DISS. ETH NO. 16839

Mastering Spam

A Multifaceted Approach with the Spamato Spam Filter System

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of

Doctor of Sciences

presented by

KENO ALBRECHT

Dipl. Inf.

born June 4, 1977

citizen of Germany

accepted on the recommendation of

Prof. Dr. Roger Wattenhofer, examiner
Prof. Dr. Gordon V. Cormack, co-examiner
Prof. Dr. Christof Fetzer, co-examiner

2006

Abstract

Email is undoubtedly one of the most important applications used to communicate over the Internet. Unfortunately, the email service lacks a crucial security mechanism: It is possible to send emails to arbitrary people without revealing one's own identity. Additionally, sending millions of emails costs virtually nothing. Hence over the past years, these characteristics have facilitated and even boosted the formation of a new business branch that advertises products and services via unsolicited bulk emails, better known as spam.

Nowadays, spam makes up more than 50% of all emails and thus has become a major vexation of the Internet experience. Although this problem has been dealt with for a long time, only little success (measured on a global scale) has been achieved so far. Fighting spam is a cat and mouse game where spammers and anti-spammers regularly beat each other with sophisticated techniques of increasing complexity. While spammers try to bypass existing spam filters, anti-spammers seek to detect and block new spamming tricks as soon as they emerge.

In this dissertation, we describe the Spamato spam filter system as a multifaceted approach to help regain a spam-free inbox. Since it is impossible to foresee future spam creation techniques, it is important to react quickly to their development. Spamato addresses this challenge in two ways. First, it has been designed to simplify the integration of multiple spam filters. By combining their different capabilities, a joint strike against spam promises the detection of more harmful messages than any individual solution could achieve. And second, we actively support collaborative spam filters that harness the collective knowledge of participating users. Such filters are therefore capable of learning about and eliminating new types of spam messages at an early stage.

Hundreds of participants use Spamato everyday. The results presented in this dissertation provide insights into real-world operating environments rather than test bed scenarios often used in other projects. The results underpin our thesis that a concerted and collaborative filtering approach is an effective weapon in the arms race against spam.

Zusammenfassung

Das Versenden von Emails ist zweifellos eine der bedeutensten Anwendungen, um über das Internet zu kommunizieren. Leider weist der Email-Dienst eine entscheidende Sicherheitslücke auf, die es ermöglicht, Emails an beliebige Personen zu verschicken, ohne die eigene Identität preiszugeben. Außerdem ist das Versenden von Nachrichten nahezu kostenlos. Diese beiden Kriterien haben in den letzten Jahren entscheidend dazu beigetragen, dass sich eine neue Branche entwickeln konnte, in der mit unerwünschten Werbe-Emails – besser bekannt als Spam – versucht wird, Produkte und Dienstleistungen zu vermarkten.

Heutzutage macht Spam mehr als 50% aller Emails aus und ist somit zu einem der größten Ärgernisse im Internet-Alltag geworden. Obwohl an diesem Problem bereits seit einiger Zeit gearbeitet wird, konnten bisher (global betrachtet) kaum sichtbare Erfolge erzielt werden. Die Spam-Bekämpfung stellt sich als eine Art Katz-und-Maus-Spiel dar, in dem Spammer und Anti-Spammer sich gegenseitig mit immer ausgefeilteren Methoden konfrontieren. Während Spammer nach Möglichkeiten suchen, um bestehende Spamfilter zu umgehen, versuchen Anti-Spammer, neue Spam-Tricks möglichst schnell zu erkennen und zu bekämpfen.

In dieser Dissertation beschreiben wir das Spamato Spamfilter-System als einen vielseitigen Ansatz, um wieder Kontrolle über die eigene Inbox zu erlangen. Da es unmöglich ist, die zukünftige Entwicklung neuer Spam-Nachrichten vorherzusehen, ist es umso wichtiger, diesen möglichst schnell entgegenwirken zu können. Spamato stellt sich dieser Herausforderung in zweierlei Hinsicht. Zum einen wurde Spamato so konzipiert, dass die Integration mehrerer Spamfilter vereinfacht wird. Die Kombination ihrer verschiedenen Fähigkeiten erlaubt ein gemeinsames Vorgehen gegen Spam, das eine höhere Erkennungsrate als die Verwendung einzelner Filter verspricht. Und zum anderen unterstützen wir kollaborative Spamfilter, die das kollektive Wissen von Benutzern zum Einsatz bringen. Solche Filter sind daher in der Lage, neue Arten von Spam-Nachrichten frühzeitig zu erkennen und zu beseitigen.

Hunderte von Benutzern verwenden Spamato regelmäßig. Die in dieser Arbeit aufgeführten Resultate geben einen guten Einblick in das reale Einsatzumfeld der Spamfilter, anstatt nur Testbett-Szenarien zu beleuchten, wie es in vielen anderen Projekten der Fall ist. Die Resultate untermauern unsere These, dass ein vereintes und kollaboratives Vorgehen gegen Spam eine effektive Maßnahme darstellt.

Acknowledgements

This thesis would not have been possible without the support of a multitude of people. First of all, I would like to thank my advisor Roger Wattenhofer for his help and guidance throughout my thesis. Starting as the "practice guy" in your group gave me the opportunity to learn that also real-world systems are usually backed by profound theories. Although our P2P game ideas never made it to a useful application, the encouraging and valuable discussions with you finally led to what became this thesis.

I would also like to acknowledge the work of my co-examiners Christof Fetzer and Gordon V. Cormack. Thanks for investing the time to read through the thesis and for the inspiring comments.

I am very grateful to all my fellow colleagues working in the Distributed Computing Group. I know that it must have been hard to domesticate one of the last Kenos living in the wild, although you never managed to control my exceptional way of ingestion (one and a half peaches is nothing) and communication (it is art, not whistling). As a long-term member of DCG, it is really hard to keep track of all the people I "should" acknowledge in this paragraph. Maybe you are so kind to condone my lack of enthusiasm to do so.

Just kidding. Of course, I would like to thank Ruedi Arnold for teaching me the weirdest Schwitzerdütsch dialect and for being a good and motivating companion all the time. It was a real pity when you left the building to find your way to *wherever* (sorry, no space for a nice picture). I would like to thank Nicola(s) Burri, who had to withstand my favorite Christmas songs 18 long months while we were working in the same office, for writing Sp$\alpha m \alpha$to and fun-in-the-office (and for constantly challenging me with nastiness); Aaron Zollinger for many interesting discussions that went far beyond Swiss cantons and spam (and for the infamous *Pipi*-paper); Fabian Kuhn for staying longer in the office than I did (and for the insight that real genius needs beer, pepper, and blick.ch); Regina O'Dell for demonstrating that job & family can be managed with ease even if not everything seems to be perfect (and for sharing my passion on food creation, meat rulez!); Pascal von Rickenbach for being a source of inspiration for all kinds of design questions (and for leaving always early so that we did not have to talk on train); Thomas Moscibroda for giving me the opportunity to mention at least one meaningful formula in this thesis: $I_\lambda \leq \frac{A(R_\lambda^+)}{A(D_i)} \cdot \frac{\theta v (3n\beta\theta^2)^{\tau(v_s)} \cdot (2r_s)^\alpha}{(((\lambda - \frac{1}{2})\mu\theta^{\frac{2}{\alpha}} - 1) r_s)^\alpha}$ (and for loving the brilliant B-boys); and Mirjam Wattenhofer for letting me play with your children although you must have known about my bad influence on minors (and for offering me a room in your flat on my very first days in Zurich).

Furthermore, I would like to extend my gratitude to Roland Flury who probably does not know how grateful everybody is about what he is doing for the group, Stefan Schmid who introduced me to Marvin, Yvonne Anne

Oswald who we never saw dancing, Thomas Locher who regularly asked me about Java problems nobody else will ever experience, Remo Meier who would arguably be able to re-write Spamato in seven days, Michael Kuhn who also likes beer, and Olga Goussevskaia who left Russia and Brazil to learn Schwitzerdütsch.

My final DCG-thanks should reach Andreas Wetzel for being the best co-admin of the Spamato project one could think of, Yves Weber for his lasting impression on our group and Spamato, and Thierry Dussuet for managing all the computer stuff much better than I ever did.

I advised numerous students throughout my PhD time whom I would like to acknowledge here for their creative work, interesting discussions, and for sharing their Swiss mindset: David Bär, Marco Cicolini, Beatrice Huber, Micha Trautweiler, Martin Meier, Thierry Bücheler, André Bayer, Stephan Schneider, Bernhard Stähli, Charly Wilhelm, Guido Frerker, Kliment Stojanov, Markus Egli, Daniel Hottinger, Till Kleisli, Fabio Lanfranchi, Roman Metz, Franziska Meyer, Lukas Oertle, and Andreas Pfenninger. Special thanks go to Gabor Cselle, your theses were probably the most interesting, challenging, and enjoyable.

My thesis would not be complete if no users ran Spamato; thanks to all anonymous participants who made my thesis a real success. I would like to express my gratitude especially to those people who used the forums to send feedback, file bug reports, or claim a missing feature; you really helped to improve Spamato. I am particular thankful for valuable discussions to Paul Coucher, Boris Považay, Jim Cairns, Dick Hoogendoorn, David Jackson, Joshua Levine, Lance W. Haverkamp, Thomas Ronayne, Volker Koch, Jörg Zieren, Gary J. Toth, and John Sanborn. And of course, my thanks extend to the students who contributed to Spamato with their theses and even afterwards: Nicolas Burri, Simon Schlachter, Andreas Wetzel, Christian Wassmer, Michelle Ackermann, Remo Meier, Markus Neidhart, Roman Fuchs, and Dennis Rietmann. Spam will have been.

Without Roderich Groß, I would probably never have started my PhD thesis in Zurich. Thanks for pointing me to Roger's job offer and for all the motivating calls and emails; now it is your turn to complete your thesis! And without Florian Schweizer (besides Aaron, Nicolas, Gabor, Thomas, Stefan, and others), I would probably never have finished my thesis, at least there would be many more grammar and spelling mistakes. Thanks for proof reading most of my thesis (and for the "Spamato tomato").

Above all, I would like to express my utmost gratitude to the whole of my family and my family "in spe." This work would not have been possible without your constant support. I am also deeply indebted to my girlfriend Lisa for her love and all the other things she definitely knows best about.

There are a lot of people whose names I just cannot remember at this moment. Due to boredom, I doubt that anyone will ever reach to this paragraph. But in case you do, please do not hesitate to insert your name here and feel acknowledged for whatever reason: ...

Contents

1 Introduction: Mastering Spam **11**

2 The Spamato Framework **15**

2.1 Related Work 17
2.2 The Filter Process 20
2.3 The Decision Maker 33
2.4 Spamato Plug-Ins 35
2.5 Spamato Add-Ons 42
2.6 Concluding Remarks 52

3 Spamato Filters **55**

3.1 Pre-Checkers 55
3.2 Bayesianato 56
3.3 Domainator 60
3.4 Ruleminator 68
3.5 Collaborative Spam Filters 70
3.6 Earlgrey 74
3.7 Razor 78
3.8 Comha 81

4 The Trooth Trust System **83**

4.1 Related Work 84
4.2 Preliminaries 85
4.3 The Trooth System 87
4.4 The Spamato Authentication and Authorization System 92
4.5 Applying Trooth to Spamato 93
4.6 Concluding Remarks 95

5 Analysis of Collaborative Spam Filters **97**
5.1 Fingerprinting Algorithms 97
5.2 Performance Criteria 99
5.3 Results for a Categorized Corpus 101
5.4 Results for the TREC Corpus 104
5.5 Concluding Remarks 105

6 The Plug-in Framework **107**
6.1 Related Work 108
6.2 The Plug-in Framework 109
6.3 Using the Plug-In Framework in the Spamato System 116
6.4 Concluding Remarks 117

7 Project Statistics and Filter Results **119**
7.1 Project History 119
7.2 Usage Statistics 120
7.3 Filter Results 122
7.4 Concluding Remarks 126

8 Conclusion **127**

Chapter 1

Introduction: Mastering Spam

```
Let me introduce you to something that is fun,
exciting and most of all enjoyable...
(April Frederick <eojutuetglubgmt@kykernel.com>, 5/27/2006)
```

Email is undoubtedly one of the Internet's killer applications. It satisfies the basic human need for communication and has become mission critical in every organization. Billions of emails are delivered each day connecting people around the globe. Unfortunately, not all emails are sent for serious purposes. More precisely, the majority of all emails circulating on the Internet are unsolicited bulk emails, in short: spam.[1]

To prevent spam from becoming email's *killer* application, a plethora of countermeasures have been proposed, for instance legal regulations, economic burdens, DNS-based attempts, and a variety of solutions exploiting different spam filtering techniques. However, the fight against spam has only been modestly successful so far: Recent studies report that currently more than 70 percent of all emails are spam, and that no improvement has been detected over the past years.

The Simple Mail Transfer Protocol (SMTP) is the root of all evil. Its authors did not foresee the danger of organized misuse, thus failing to devise a mechanism to prevent the flooding of millions of inboxes. Particularly, the lack of reasonable authentication schemes enables spammers to operate incognito. Proprietary mechanisms, such as the Sender ID Framework and the DomainKeys system, promise to help alleviating this deficiency, but it might take years before they are widely deployed and adopted.

[1] There is no standard definition for the term *spam*. In this thesis, we adopt the definition by Cormack and Lynam [22]: Spam is "Unsolicited, unwanted email that was sent indiscriminately, directly or indirectly, by a sender having no current relationship with the recipient."

A significant improvement of the current situation can only be achieved on a global scale. While governments around the world are slowly beginning to enact and enforce laws for punishing spammers, legal regulations suffer from national limitations. Although known mass spammers like Wayne Mansfield have been committed to prison in Australia, similar business in China or Russia are still considered legal. Even the U.S. *CAN-SPAM Act*, one of the first anti-spam laws in the world, has not effected a decline in the number of companies spamming from the United States' own territory.

We believe that no panacea exists to remedy the spam pest. Probably, even combinations of the aforementioned solutions will not eliminate spam in the near future. If one assumes that spammers will *always* be able to advertise their products by email, the primary task is to prevent people from reading these messages.

In this thesis, we describe the *Spamato spam filter system* as a multifaceted solution to the spam vexation. While spam filters do not combat the creation of bulk emails, they avoid the task of manually separating spam from legitimate (ham) messages. However, a single filter can fail when spammers accomplish to conceal their real intentions. As two heads are better than one, Spamato's basic intention is to bundle a variety of complementary spam filters, which concertedly seek to master the ever increasing flood of spam messages. At first glance, implementing a spam filter system might seem an easy task: receive an email and return whether it is ham or spam. But that is by far not the whole story: "Where does the message actually come from?", "How can filter results be combined?", and "What to do with the overall classification?" are only a few questions that we will answer starting in Chapter 2.

In Chapter 3, we will take a closer look at the six default Spamato filters. Among them are three *collaborative spam filters* that harness the collective knowledge of spam fighting communities. In a nutshell, if one user denounces an email as spam, collaborative spam filters remove *similar* emails from other inboxes as well. However, such filters are prone to manipulation if implemented without caution. For example, imagine the discontent that would be caused if someone reported common newsletters as spam to the community. Such scenarios are prevented by the Trooth trust system, which will be detailed in Chapter 4.

In Chapter 5, we will evaluate the capability of collaborative spam filters to match similar emails. For this purpose, we define several performance criteria that we use to compare the filters when tested on two different email corpora. For the first corpus, we manually categorized spam emails into classes of similar emails. The second, flat corpus is used to verify the results regarding misclassifications among ham and spam emails.

To support the multifaceted nature of Spamato, we provide a software architecture that is extendable in several respects. For instance, additional

filters or tools, such as whitelists, can easily be plugged into our system to improve the accuracy of the entire system. The foundation of this extendability is based on the plug-in framework presented in Chapter 6.

Spamato is a real-world project running out-of-the-box on arbitrary desktop machines. It is written in Java and seamlessly integrates into several email clients, such as Thunderbird and Outlook. We will highlight some of the project's milestones in Chapter 7 and also present usage statistics. Furthermore, we will discuss the results of the Spamato filters. The analyzed data was taken from our statistics server, which stores information about detected spam messages of hundreds of users.

Finally, we will conclude this thesis in Chapter 8, where we will also give a brief outlook on future research directions.

Chapter 2

The Spamato Framework

`Are you perhaps interested in buying SPAMATO.COM ?`
`(Jessy Timber <j.timber@yahoo.com>, 12/26/2005)`

Spamato is a *spam filter framework* [3]. It does not filter any emails on its own; it even lacks the capability to distinguish between spam and ham messages. The main purpose of Spamato is to establish an extendable system that allows the bundling of several components that detect spam in concert. Spamato provides only the bare framework for this task; additional plug-ins—the tools, nuts, and bolts of the system—do the real job. Figure 2.1 gives a rough overview of the Spamato architecture and its interacting components. In this chapter, we present the design decisions that underlie the Spamato framework and discuss its central concepts.

A bare framework is of no use without those nuts and bolts. One key characteristic of Spamato is its extendability—the capability to fill the empty space in the framework with building blocks called *plug-ins*. The support of several independent spam filters as plug-ins for the system was a driving consideration when designing Spamato. We provide information about the filters to be bundled with the framework when deploying the Spamato system as a whole in Chapter 3. In Chapter 6, we describe the plug-in framework which lays the foundations for Spamato and can be used for other applications as well.

One question that arises when dealing with more than one spam filter concerns the order in which they should be processed. We could run them one at a time. However, if the processing time of different filters varies significantly, we might end up waiting a great amount of time until we get all their responses. We could run all of them concurrently. But if the effectiveness of filters differs, invoking those with greater impact first seems more to the point. The Spamato framework does not define the filtering order; it is the task of a *filter process* component to define it and to activate the filters in this order. Based on different parameters such as the runtime, the effectiveness,

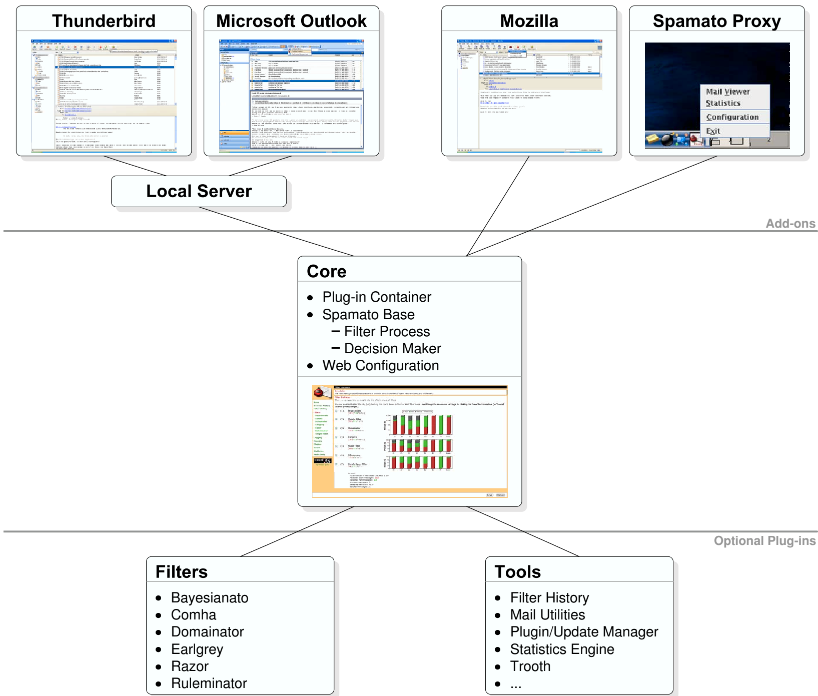

Figure 2.1: A rough overview of Spamato's architecture: The Spamato *Core* filters emails received from *Add-ons* with the help of spam *Filters* and (optional) *Plug-ins*.

and any other related information, it is capable of laying out the ordering in which filters will check messages intelligently.

The combination of different spam filters has many advantages, as several filters can catch more spam than a single one. One challenge, though, when working with two or more filters is how to combine their results. For instance, imagine a situation where one filter draws the conclusion that an email is spam and another filter thinks it looks legitimate. Is it spam or ham? In this case, we might also have to determine if all filters are equally effective or if we should assign more value to specific results. Spamato does not provide an answer to these questions. But it provides the means to let others decide: A *decision maker* can be plugged into the framework to draw the final conclusion based on the result of each single filter.

Obviously, the concepts of filter processes and decision makers are related. As we will see later, the filter process component calls the decision maker

whenever an email has to be classified. In Sections 2.2 and 2.3, we detail these concepts and show how they are combined to filter spam.

Spam filters, the filter process, and the decision maker are the most important components in the Spamato framework. When given an email, they decide how to classify it—to tell the good from the bad. But Spamato is not restricted to these components. In fact, the number of other plug-ins in the Spamato system is greater than the number of spam filters. Although these plug-ins do not contribute to the effectiveness of spam filtering, they improve the overall usability, thereby enhancing the user and the developer experience. For instance, the *filter history* component presents the filter results in a way that helps understand the filter process. By doing this, it answers the question "Why has a specific message been classified as spam?"—an interesting piece of information for both users and developers. In Section 2.4, we present some of the additional plug-ins and show how they are integrated in the Spamato framework.

So far, we have suggested that Spamato can classify emails only: We have neither addressed the question how these emails are delivered to Spamato nor what to do about the results. In fact, Spamato does not care! But it is obvious that a user does care. To reconcile these conflicting views, *add-ons* are connecting Spamato with the source of an email. Our preferred source is an email client, such as Outlook or Thunderbird, which delegates an email to Spamato and presents the result to the user, for instance by highlighting or moving detected spam messages to a special "junk" folder. We detail this aspect in Section 2.5, where we also describe alternative approaches.

As we will see in the next sections, Spamato is extendable in many ways. Its current purpose is to filter spam—hence its name. On the other hand, the framework we describe here can also be useful to enhance the handling of email in general. We sketch some of our future ideas in Section 2.6, presenting the *Emailato*.

A final purpose of Spamato is to encourage other developers to implement their own spam filters and arbitrary plug-ins. We also like to see Spamato ported to email clients other than those that are already supported. We provide an open-source framework which makes it easy to enhance it with new technologies. Developers are relieved of the burden to re-invent the wheel for fundamental operations such as parsing emails or analyzing results. Spamato provides the general environment of a viable anti-spam solution and controls the final deployment.

2.1 Related Work

SpamGuru is a server-side spam filter system incorporating several email classifiers [88]. It is a commercial, closed-source project developed by IBM and is integrated with the Lotus Note Suite, which provides a feedback channel

for the users. The authors note that future releases will contain a public API so that third-party and user-provided classifiers can be supported. However, it is unclear in which form developers will be able to do so.

SpamGuru uses a highly optimized "filter pipeline," in which its filters are ordered to maximize the message throughput of a server. In the pipeline process, each filter assigns a score to the email checked, which can be positive to indicate the spamminess of an email and negative for ham. Once the score exceeds a spam threshold, the filter process is stopped and the mail is handled as spam, for instance marked as such and forwarded to the client. Similarly, if the score falls below a ham threshold, it is immediately handled accordingly. Messages whose score is somewhere in between can be marked as borderline cases.

The pipeline process routes emails through filters of increasing complexity and cost. It starts with simple whitelist or blacklist tests while more elaborate filters, such as the Chung-Kwei [78] pattern-based algorithm, are invoked last. A final linear *super-classifier* is used to combine the weighted results of the individual filters. By experimentation, it was found that even the combination of two classifiers can be beneficial.

SpamGuru exhibits a very sophisticated filtering approach. In contrast to Spamato, though, it seems to be a rather monolithic block whereas Spamato is much more flexible. Furthermore, Spamato is mainly intended to run as a client-side spam filter and its architecture allows integrating it with different email clients. On the other hand, we could improve our decision maker and adopt some of the SpamGuru's ideas regarding a linear super-classifier.

SpamPal is an open-source SourceForge project licensed under the GPL [110]. It is an email proxy for Windows users supporting POP and IMAP accounts. It has been written in C and provides an extension interface to add third-party plug-ins as Windows DLLs. So far, over 30 plug-ins have been contributed to the project, which is utilized by approximately 50000 users.

As in SpamGuru, an email is routed through a filter pipeline. But in contrast to SpamGuru, each filter can flag an email either as definite ham or spam; no scores are assigned. In fact, the developer's guide of SpamPal states that once the "spam" flag has been set, subsequent classifiers should not further check an email. This clearly contrasts the approach we have motivated for Spamato, where we want to employ as many filters as possible (or at least as necessary). It also shows the inflexibility of SpamPal's filter process and the simplicity of its decision maker.

Although we assess the extension mechanism of SpamPal to be inferior compared to ours, it also allows for the addition of arbitrary filters and utility plug-ins. Several extension points are provided by the SpamPal framework, for instance to filter only detected domains or header information. Furthermore, information assembled by one filter can be stored in a *session object* to be used by other filters as well. Finally, also a default configuration mech-

anism has been designed, which is similar to what is provided by the plug-in framework we use for Spamato.

SpamPal is an email proxy limited to the Windows environment. In contrast, Spamato cannot only be used as a proxy but can also directly be integrated into email clients, which is more user-friendly especially for receiving feedback. Furthermore, as Spamato is written in Java, it is portable to other operating systems besides Windows. However, SpamPal's proxy and plug-in architecture allows for filter types which Spamato currently does not support. For instance, SpamPal is able to download and check only the headers of an email, while filters in the Spamato system are always provided with the complete email.

SpamAssassin is probably the most widely used spam filter system with millions of installations [86][140]. It is an open-source project hosted by the Apache group, written in Perl, and available for several platforms. Usually, it is directly combined with an SMTP server or invoked from email processing tools such as Procmail; client-side proxy installations are possible but rare.

SpamAssassin literally contains hundreds of spam (and ham) tests. And it can easily be extended with custom regular expressions or more sophisticated Perl code. When checking an email, each rule assigns a positive (spam) or negative (ham) score to it. If the final score exceeds a given threshold, it is deemed to be spam and tagged as such. The default scores for each rule have been determined using a fast "Perceptron Learner" on a large spam and ham email corpus [26]. The scores can be adjusted manually or by retraining them on user-specific email archives.

SpamAssassin is a professional spam filter system that is intended to be maintained by a trained administrator rather than by a desktop PC user. In contrast, Spamato tries to provide similar capabilities on the client-side in a user-friendly manner. For instance, Spamato filters can be configured using a web browser interface whereas in SpamAssassin one has to edit plain-text files. Moreover, SpamAssassin provides only a rudimentary feedback channel: Emails can be reported or revoked by running a command line tool for which the email has to be stored in a text file. In the Spamato system, emails can comfortably be reported by clicking a button.

Using Spamato in combination with SpamAssassin seems an interesting option. Many companies use SpamAssassin on their email servers and add some header lines to the email that contain the filtering result. We utilize this result as a spam indicator in the Ruleminator (see Section 3.4), relying thus on information provided by SpamAssassin. An alternative approach, which we might look at in the future, is to invoke SpamAssassin in the normal filter process.

The Email Mining Toolkit (EMT) [91] is a data mining system that is used to compute behavior profiles and email models based on user email accounts. In addition, it has been extended to filter spam as well. In the

filter process, several machine learning algorithms are employed. A meta-classifier combines their results to reduce the overall misclassification rate [44].

Although the EMT comprises different spam filtering techniques and can be extended with similar ones, it has not been designed as a pluggable system supporting arbitrary filters. The filter process does not allow for easy modification of the ordering of filters, and the overall system is rather static as opposed to Spamato's flexibility. Furthermore, it is unclear how the proposed meta-classifier algorithms would perform in a system with many filters whose results are just binary spam/ham decisions rather than confidence intervals. Nonetheless, it would be interesting to combine their filter system with Spamato, which might be possible as both are written in Java.

The EMT has first been used to analyze offline email archives. The Profiling Email Toolkit (PET) is a wrapper for the EMT that allows for online statistics; an extension for the Thunderbird email client, which handles all incoming messages, is described in [43]. The idea of the PET is similar to our add-on approach. But in contrast to the PET, we have given proof of its compatibility with other email clients.

There is a variety of other spam filter systems which combine multiple spam filters in order to approach spam in a multifaceted way. But since they usually do not provide any extension mechanisms, they are not comparable to what Spamato aims for.

2.2 The Filter Process

In the filter process, all spam filters are invoked to check emails. The combination of the classifications of all filters results in a spam or ham decision. The structure of the filter process is not fixed by the Spamato framework. It can be modified to meet different requirements, such as the time it takes to filter a message, the resources used to achieve this, or the consideration of different capabilities of spam filters. The requirements themselves depend on several properties, such as the environment in which Spamato is being used, the number of emails received daily, or simply personal preferences. For instance, considering resource consumption is much more important on the server side than on the client side: A server might have to check several thousands or even millions of emails per day, a client only tens or a few hundreds. Just like the selection of the right number of neurons or perceptrons in a neural network or its hierarchy is important to adapt to changes in its environment, an effective anti-spam solution depends on the number of components, their properties, and their ordering in its filter process.

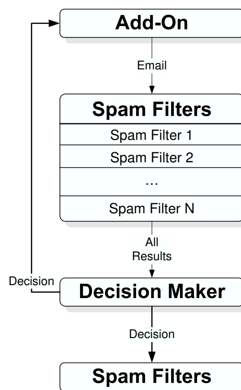

Figure 2.2: In the *Simple Filter Process*, each spam filter checks an email one after the other. All results are evaluated in a decision maker which returns the overall decision to the add-on and also notifies all spam filters.

In Section 2.2.1, we describe our first approach of a filter process component that had no real requirement other than to work. With its simple structure, it was performing well for a small number of similar filters and few emails to be checked. But it lacked performance and flexibility where these conditions were not met.

Our second approach is described in Section 2.2.2. Here, we considered the different running times of filters and implemented a new concept of *pre-checkers*—spam or rather ham filters that definitely know when they face a legitimate email. Thus, we introduced a kind of *layering* in the filter process, where filters are ordered depending on their properties and their types. We also added *post-checkers* to our filter process, which are notified with the final decision of the filter process.

The third approach is a future project as it has not been released yet. As outlined in Section 2.2.3, it takes the idea of layering even further. By arranging not only pre-checkers and spam filters but also *pre-processors* and explicit *filter phases*, the filter process gains an additional level of flexibility that leads to faster results while using less resources. For example, a pre-processor can calculate data, such as URLs contained in an email, for several subsequent filters, thereby avoiding redundant work. And very fast filters that check the header information of an email only could be invoked before collaborative filters query a server.

So far, we have talked about the filter process. When an email is manually reported or revoked by the user, a simpler approach can be used. We shortly describe how to handle reports and revokes in Section 2.2.5.

2.2.1 A Simple Filter Process

The structure of the *Simple Filter Process* (*SiFiP*) is depicted in Figure 2.2. In the SiFiP, emails are processed in three steps:

(1) When an email arrives, a new filter process is initiated. The email is checked by all spam filters consecutively.

(2) After all filters have checked the email, all results, that is whether a filter result is positive or negative, are sent to the decision maker. The decision maker then calculates the overall decision.

(3) The decision maker publishes the overall decision to two destinations:

- The add-on receives the decision and usually moves spam messages to a special junk folder.
- All filters receive the decision and can adapt to the result. For instance, a Bayesian-based filter can update its token table.

This approach is rather simple and has several weaknesses. First, the number of concurrent checks is unbounded. That is, whenever a new email arrives also a new filter process is started. Consequently, the overall resource consumption is unbounded and the use of this filter process might lead to a shortage of memory or CPU power. Second, the decision maker is activated not until all filter results are available. A very slow filter, for instance a collaborative one that suffers from a slow network connection to a server, can protract the whole process. Third, only the add-on and all spam filters are notified of the overall decision. Other plug-ins, such as the statistics engine or the filter history component, which are also interested in the decisions, have to "disguise" themselves as spam filters in order to get the necessary information. And finally, all filters are considered to be equal; they can decide on "spam" or "ham" only (but not on "definitely ham" or "probably spam"). While this is not a particular problem of the filter process but of the definition of the capabilities of spam filters in general, the filter process would have to manage more elaborate spam filters which the SiFiP cannot do.

We will take a closer look at the filter process when we tackle these problems in the next section. We will also define some general concepts that we have identified in the lessons learned here.

2.2.2 The Current Filter Process

We have designed our *Current Filter Process* (*CuFiP*) to satisfy new requirements and generalize the earlier approach. We introduced two new concepts: A *pre-checker* is a special kind of spam filter that can *veto* in order to stop the filter process early. They are often referred to as "whitelist" components,

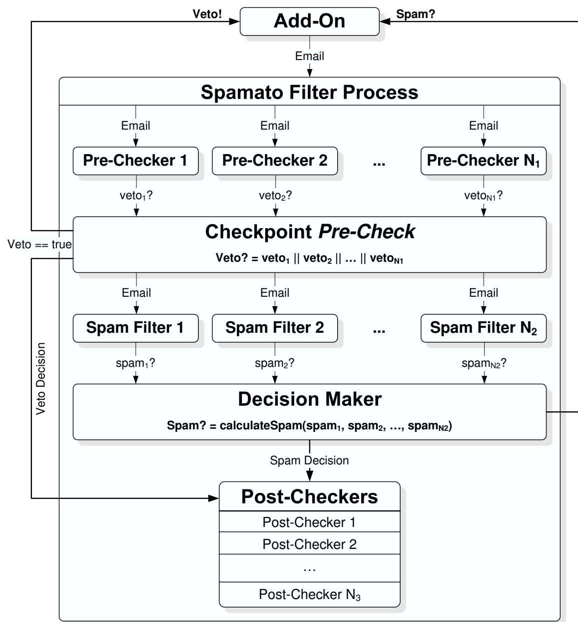

Figure 2.3: Our current filter process is structured as follows: After an email has arrived, pre-checkers can veto against its further processing. Then, all spam filters check the email concurrently. The final decision is determined by a decision maker, which publishes the outcome to all registered post-checkers.

as they definitely consider an email to be ham; further spam filters are not employed. Note that pre-checkers are complementary to spam filters as they detect ham rather than spam. We have introduced this concept mainly to reduce the number of false positives, as it is not necessary to check obvious ham emails by spam filters. In contrast to pre-checkers, *post-checkers* are called *after* an email has been classified. It is provided with the final decision and can process it further, for instance to learn from the outcome or to update statistics.

The CuFiP is illustrated in Figure 2.3. It passes emails through the following five phases:

(1) When an email arrives, a new filter process is initiated. First, the email is concurrently checked by all pre-checkers.

(2) If one of the pre-checkers vetoes against further processing, that is if one of them can definitely classify the message as ham, we return this result to the add-on and proceed with step 5.

(3) Otherwise, the email is concurrently checked by all spam filters.

(4) After all filters have checked the email, all results, that is whether a filter considers a message to be spam or ham, are sent to the decision maker. The decision maker then calculates the overall decision.

(5) The decision maker sends the overall decision to two destinations:

- The add-on receives the decision and usually moves spam messages to a special junk folder.
- Post-checkers are notified about the outcome for further processing.

This approach improves over the SiFiP in several ways. First, pre-checkers offer an important feature that allows for example integrating sender whitelists: Emails from trusted senders do not have to be examined. Although pre-checkers are similar to spam filters voting for ham only, there is an additional constraint: They have to be light-weight in terms of their resource consumption and should be very fast. The whole pre-check phase is completed within milliseconds whereas the full filter process can take several seconds. Of course, this depends on the actual implementations of pre-checkers and spam filters. But besides helping to improve the filtering accuracy by selecting definite ham messages, pre-checkers also help to reduce the resources that would be necessary when scrutinizing emails thoroughly.

Another asset of this filter process is the explicit identification of post-checkers. The SiFiP contains this concept implicitly by notifying all spam filters of the decision maker's result. Here, we describe this concept in a general form allowing any plug-in to register for decision events using the extension mechanism. All plug-ins, including spam filters, might (but do not have to) be post-checkers and can thus analyze and respond to the final decision.

In the previous section, we have criticized the SiFiP mainly in four points: the problem of an unbounded number of concurrent emails being checked, the fact that all filter results are necessary to derive the final decision, the problem that plug-ins have to be "in disguise" to be notified of decision events, and that all filters are handled equally. The latter two points of criticism have been solved by introducing pre-checkers and post-checkers. We have addressed the other two problems as follows:

To solve the first issue, we use a thread pool with a limited number of threads in the filter process. That is, we allow only a specific number of emails to be checked concurrently. This gives some control over the resources used and can easily be adjusted to different environments. A simple but effective technique.

Solving the second problem is more difficult because one has to determine a decision before all filters have calculated their result. To accomplish this,

we need a decision maker that can classify an email having incomplete information only. In the CuFiP, all filters start checking an email at (roughly) the same time. As filters vary in their processing time, they will consequently finish their tasks at different points in time. We have a decision maker classify an email whenever a new spam filter result is available. The decision maker can determine a *pre-decision* when satisfactory results are collected. For example, if 6 out of 10 filters have voted for spam, a decision maker could pre-decide on spam without regarding further filter results; similarly, if a large number of all filters have voted for ham, a preliminary ham decision cannot be questioned by outstanding spam results.

A pre-decision is defined to be final once it has been calculated. The associated decision maker will not change its mind even if more spam or ham results become available. Thus, a pre-decision can be returned to the user's email client and increases the filtering speed experienced by the user.

Of course, not all decision makers are able to calculate pre-decisions, as this depends on their algorithms. Also, the processing time and accuracy of spam filters influence the capability to derive quick pre-decisions. We show in Section 2.2.4 that the overall gain in terms of responsiveness can be significant.

However, there is one downside to this approach: We do not want to—or even cannot—always stop filters even though we might not need their results anymore. This has two reasons. On the one hand, we are interested in all filter results to analyze their behavior. We use this information for statistical reasons and also to learn which filters perform best. The latter is specifically of interest to build better decision makers that rely on previous results. On the other hand, from a technical point of view, if a filter does not react to a "we-don't-need-you" signal, at least in Java there is just no mechanism to actually stop it. That means we can notify the add-on of the pre-decision, but we still have to wait until the last filter finishes, that is after an email has completely been checked.

2.2.3 A Future Filter Process

In this section, we develop some ideas for a *Future Filter Process* (*FuFiP*), which we plan to release in version 1.0 of Spamato.

First, we introduce *pre-processors* as plug-ins that prepare information, such as the URLs found in an email or an HTML-free version of its body, for subsequent components in the filter process. Pre-processors run before any filters are called so that the latter can rely on data calculated by pre-processors. Thus, redundant work is reduced and the filtering speed is increased.

Next, we rename pre-checkers to *ham-checkers* and introduce their analogue as *spam-checkers*. Spam-checkers are spam filters that definitely know when a message is spam; they provide the possibility to "blacklist" emails.

For example, in a company, often server-side spam filters exist which flag messages before they arrive in a user's inbox. Spam-checkers are now able to sort out these "pre-checked" emails without scrutinizing them in the (probably expensive) spam filter phase. Although this concept conflicts with our credo that many spam filters are better than one, we believe that in some cases this approach is not only justifiable but also preferable. The subsequent spam filter phase remains in which a decision maker determines the final outcome.

Finally, post-checkers are renamed to *post-processors* to comply with our new naming scheme. Additionally, post-processors can analyze not only the final decision but all information that has been calculated by all components in the filter process.

We describe several pre-processors and post-processors in Section 2.4. Since some filters also integrate these notions, we will turn our attention to this topic again in Section 3.

The development of our filter processes has lead to a finely grained layering. We have started with spam filters only in the SiFiP, added pre-checkers and post-checkers in the CuFiP, and injected pre-processor and spam-checker layers now. Continuing this evolution consistently, we introduce a general filter process as shown in Figure 2.4.

The FuFiP contains an arbitrary number of *filter phases* connected in series. A filter phase bundles several filters and a decision maker that calculates the decision for this phase. The entire filter process stops when one of the decision makers calculates a final decision or the last phase has finished. The assumption is that filters and decision makers are different in each phase. That is, using our former terminology, the decision maker in the ham-checker phase concludes "ham" if one of its associated filters classifies a message to be ham, the decision maker in the spam-checker phase evaluates to "spam" if one filter votes for spam, and the decision maker in the spam filter phase evaluates to "spam" or "ham" based on its inherent algorithm. In fact, the whole filter process can be regarded as a filter phase that calls "sub filter processes," or in other words: filter processes (or phases) can recursively be combined.

The FuFiP can now easily be modified to integrate further layers without changing its structure. Accordingly, only new filter/decision maker components need to be defined. Additionally, the order in which the phases have to be run is fixed. Within each phase, filters and decision makers may be changed without influencing other phases.

Further improvements

Currently, we assume that spam filters are content filters. Spamato delivers the complete mail including its headers and body part to all spam filters

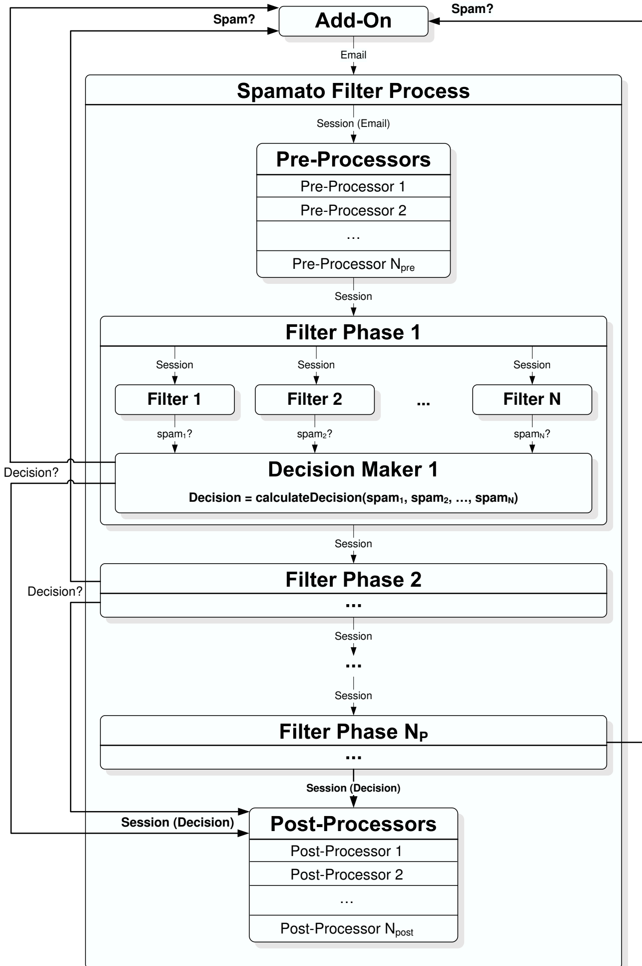

Figure 2.4: The future filter process connects several filter phases in series. In each phase, a decision maker evaluates the results of filters. If a decision can be determined, the filter process stops and returns the decision to the add-on. Otherwise, it proceeds with the next filter phase. Pre-processors can calculate data for subsequent filters to decrease redundant work. Post-processors are similar to post-checkers and can analyze the final decision.

regardless of what kind of check they perform. One idea is to split filters into groups of header filters and content filters and call header filters in a separate filter process first. This would be advantageous in two respects: First, fewer resources are used, as only a small part of the message has to be downloaded. Additionally, header filters are usually faster, as they perform only simple analysis or regular expression matches. And second, this prevents from downloading possible harmful attachments containing viruses to the user's machine.

We have motivated the concept of filter processes with the argument that it can structure the ordering of the spam filters. So far, we have tackled this problem by introducing the layering architecture for a hierarchical, type-based ordering. Also, thread pools are used to call spam filters concurrently. A further idea is to order spam filters depending on their processing time and their effectiveness. For instance, if it were possible to get the most significant filter results first, a ham or spam pre-decision could prevent the launching of time-consuming collaborative filters. We have touched upon this topic before, but in the FuFiP it is possible to order filters within each filter phase separately—introducing an even more powerful mechanism to tune the whole filter process.

A downside, though, is that implementing these ideas increases the complexity of the filter process; the management overhead for users and developers might outweigh their benefits. Therefore, we have decided to exclude these features in the next Spamato release and to keep them for future work.

2.2.4 Comparison of the SiFiP and the CuFiP

We have conducted two experiments to prove the superiority of the current over the simple filter process. We were particularly interested in what benefits are derived from the use of pre-checkers and pre-decisions as introduced in the CuFiP above in terms of saved processing time. We cannot show the advantages of the FuFiP empirically since the rationale for this approach was to make a design decision to support managing complex filter processes. We also have not implemented any spam-checkers yet, but we are confident that their employment would bear a similar gain as we show here for pre- (or ham-)checkers.

For the experiments, we have collected usage data of ten voluntary participants over a period of several weeks. The surveyed data provide information about the total number of ham and spam messages users received and about the time Spamato needed to process these emails. For privacy reasons, we usually do not collect this kind of detailed information. Thus, we have not been able to compile such data from all Spamato users in our statistics database. The users were running Spamato with different add-ons (see Section 2.5), email clients, computers, and operating systems. Nonetheless, we think that this has only little impact on the results presented here.

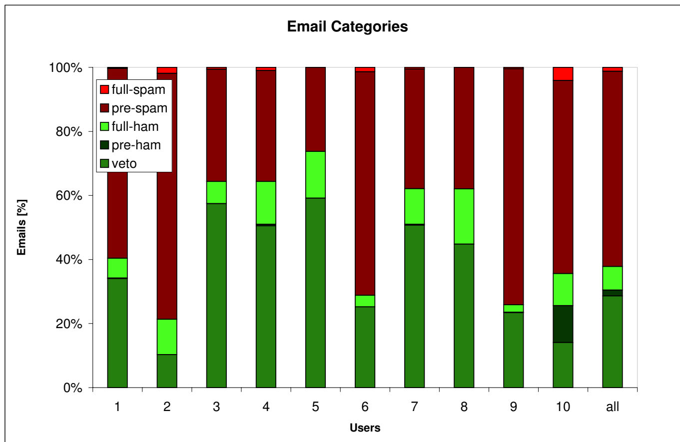

Figure 2.5: Distribution of email categories. This overview reflects the general impact of pre-checkers and pre-decisions, as both shorten the processing time of emails. The emails of 10 users are classified into different ham and spam categories: veto emails have been detected by a pre-checker, pre-ham and pre-spam emails have been classified as a pre-decision, and full-ham and full-spam emails have been checked by all filters before classification.

Collected Data

We have divided the legitimate emails into three groups: emails that have been detected by a pre-checker (*veto* emails), emails that have been detected before all filters have processed it (*pre-ham* emails), and emails that have been classified as ham only after being processed by all filters (*full-ham* emails). Similarly, we categorize spam emails into *pre-spam* emails and *full-spam* emails; note that there is no spam complement to veto emails. False positives and false negatives are not considered here. That is, in our analysis, false positives are spam and false negatives are ham messages.[1] Unless otherwise noted, all values were determined using a min-spam decision maker (see Section 2.3.1) with the default min-spam value of 2.

Figure 2.5 shows the number of emails per category and user (normalized). For example, 40.4% of all emails of User 1 are ham emails: 34.1% veto, 6.2% full-ham, and very few pre-ham emails. Similarly, 59.6% are spam messages: 59.3% have been detected as pre-spam and 0.3% as full-spam messages. In total, there are 11036 messages to which User 1 contributed the largest number of 4877 messages and User 8 only 145. 37.8% of all emails are

[1] The actual number of false negatives and false positives was insignificantly low. Thus, dropping this kind of information does not influence the overall result.

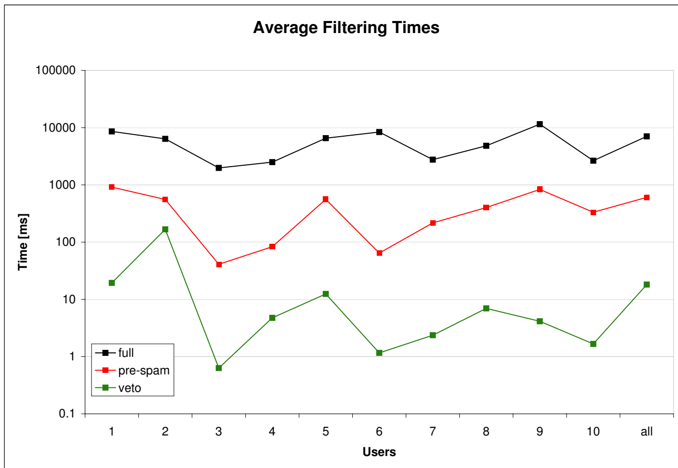

Figure 2.6: This chart provides information about the average processing time of emails separated by categories and users; note the logarithmic scale of the time axis. Generally, veto emails can be detected within few milliseconds, pre-spam messages are detected within the first second, and a full check performed by all filters costs up to 10 seconds of processing time.

ham (28.6% veto, 1.8% pre-ham, 7.4% full-ham) and 62.2% are spam emails (61% pre-spam, 1.2% full-spam).

It is noteworthy that, on average, about 76% of all legitimate emails were detected by a pre-checker. Also, almost no spam email has to be checked by all filters since a pre-decision sorts out such messages earlier in the filter process.

Figure 2.6 shows the average processing times of the filter process per email category and user; note the logarithmic scale on the time axis. For this chart, we do not consider pre-ham results because their occurrence is too low. Furthermore, the "full" times is for both full-ham as well as full-spam emails. For example, for User 1, veto decisions have been made on average within 19 milliseconds, spam pre-decisions within 914 ms, and processing all filters took about 8.6 seconds. For all users, on average, veto decisions took 18 milliseconds, spam pre-decisions 605 ms, and processing all filters took about 7.1 seconds. Note that the high "full" times are dominated by the processing time of collaborative filters, such as the Earlgrey or Razor filter (see Chapter 3); mainly, unpredictable network latency and server load can lead to these extreme values.

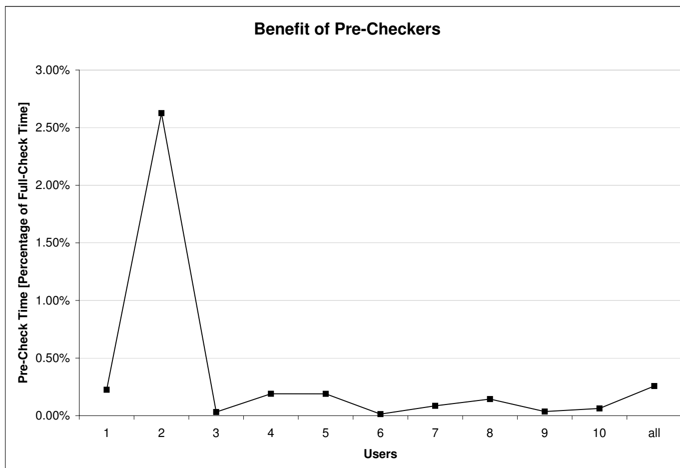

Figure 2.7: The benefit of pre-checkers is calculated as the ratio between the average veto and the full-check times. As can be seen, for most users a pre-checker takes less than 0.5% of the processing time a full check would usually take. That is, it decreases the average time spent on processing a ham email by a factor larger than 200.

On the Benefit of Pre-Checkers (Fast Ham Decisions)

From the given numbers, we can calculate the benefit gained when employing pre-checkers rather than performing full checks. We express this benefit as the saved time by dividing the average time of pre-checks (vetoes) by the average time of full checks. This benefit is depicted in Figure 2.7 for each user.

For example, the value for User 1 means that, on average, a pre-check takes only about 0.23% of a full check. That is, a pre-check speeds up the processing of ham emails for User 1 by a factor of about 450. Except for User 2, whose average pre-check time is higher than for the others, all values are well below 0.5%. Averaged over all users, a positive pre-check is almost 400 times faster than a full check; instead of 7 seconds, it takes only 18 milliseconds. Although we have not measured the impact of spam-checkers as described in the FuFiP, we believe that they could provide a similar gain.

On the Benefit of Pre-Decisions (Fast Spam Decisions)

Using pre-decisions when the overall result of the filter process can be settled without waiting for further filter results also speeds up the filter time realized

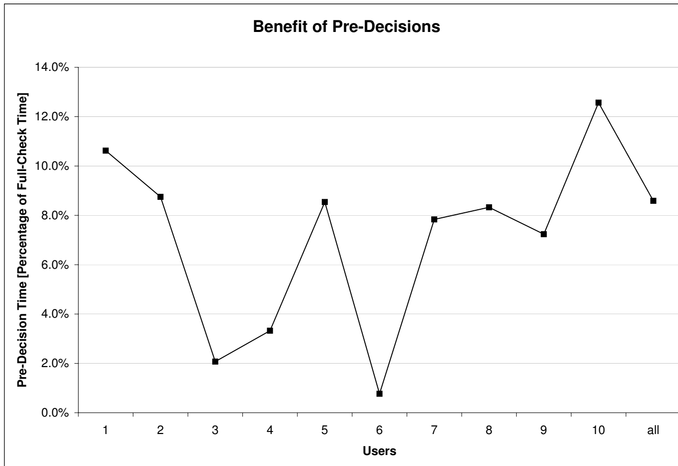

Figure 2.8: The benefit of pre-decisions is calculated as the ratio between the average pre-spam and full-check times. As can be seen, a spam pre-decision takes only between 1% and 13% of the processing time a full check usually needs.

by users. We establish the benefit as in the previous section by dividing the average time of pre-decisions by the average time of full checks. The benefit is depicted in Figure 2.8 for each user.

For example, the value of User 1 means that, on average, a spam pre-decision can be made in about 10.6% of the time necessary for a full check. In other words, the speed up factor is approximately 10. The values differ between 0.8% and 12.6%. Averaged over all users, the value is about 8.6%.

2.2.5 The Report and Revoke Process

The user can manually report and revoke emails which have been misclassified. A *report* is when a user labels a message to be spam and a *revoke* when a user labels a message to be ham. Reports are usually associated with false negatives—emails that have not been detected as spam by the filter process. Analogously, a user revokes a false positive whenever a legitimate email has wrongly been classified as spam.

Reports and revokes build an important feedback mechanism that helps to recall and rectify previous decisions. Additionally, it helps to train (machine-) learning filters, such as Bayesian-based ones, and provides collaborative filters with data for their databases. As the handling of reports and revokes is similar, we will use the term “reports” for both reports and revokes in the remainder of this section.

The report process is simple compared to the filter process. All spam filters are provided with reported emails in sequence. It is not necessary to process user feedback concurrently, as there is no need to have them processed fast. In fact, we put emails into a queue and have a single consumer thread handle them. In addition to filters, *report-processors* (and *revoke-processors*) can be registered to be notified of these events. Such plug-ins include for example the statistics engine and the filter history, which we describe later in Section 2.4.

Despite the simplicity of the report process, there is one challenge: A reported email has to be handled after the filter process has announced its final decision (if there is any). With pre-decisions it is possible that reports arrive from a fast reactive user before the final decision has been determined. They would thus be handled in the wrong order, which would be counter-productive. For example, imagine the case in which a legitimate email has been determined to be spam. As a pre-decision, the wrong result is sent to the email client before all filters are finished. A fast reacting user corrects the mistake and revokes the email. Now, the revoke event is sent to a Bayesian-filter which treats the tokens in the email to be ham tokens. After all filters have checked the email, the final decision, which is still spam, as it cannot differ from the value of the pre-decision, is also handled by the Bayesian-filter—but now the tokens are considered to be spam. This example shows why the ordering of final decisions and report events matters and that it is therefore important to be guaranteed.

2.3 The Decision Maker

The decision maker is an important building block in the Spamato framework. The plug-in architecture allows modifying its implementation easily. The general task of a decision maker is to aggregate several spam filter results in order to determine the overall classification of an email. The classification is usually "ham" for legitimate emails or "spam" for unsolicited emails. It can also be "unknown" if the number of available filter results is too low or the results are too ambiguous to decide for the one or the other. The main goal of a decision maker is to detect as many spam messages as possible (true positives) while minimizing the number of false positives.

We have experimented with several decision makers before implementing the simple but effective *Min-Spam Decision Maker*, which is described in the next section. It works well for a small number of distinct and powerful spam filters like those deployed with the current version of Spamato. However, with a growing number of more light-weight spam filters, for instance by creating rules with the Ruleminator testing for common but imprecise spam indicators, other approaches might be more appropriate; we present some ideas in Section 2.3.2.

2.3.1 The Min-Spam Decision Maker

Our current *Min-Spam Decision Maker* (*MiSDeM*) is rather simple: It allows defining the minimal number of results that have to be "spam" in order to classify a message as such. For example, a min-spam value of 1 means that only one spam filter has to consider an email to be "spam" to have an overall classification of "spam." This means to accept every single "spam" result as granted—an approach that would make sense only if each spam filter had a very low false positive rate. In fact, a min-spam value of 1 can be used in the spam-checker phase, where exactly this behavior is desired (see Section 2.2.3). Also, a similar approach with a *min-ham* value of 1 is used in the pre-checker phase of the CuFiP as described in Section 2.2.2. For the general spam filter phase, though, it is definitely not appropriate to select such a low value.

To achieve different accuracy levels, the value can be adjusted to meet a user's expectation. As Spamato is a multi-filter system that tries to take advantage of the combined capabilities of several filters, we have chosen a default min-spam value of 2. Surprisingly, although this value is still very low, it has proved to be quite effective for most users running the default Spamato system.

The MiSDeM allows for pre-decisions and thus meets one of the requirements for a fast filter process as described in Section 2.2. Whenever the min-spam criterion is fulfilled, that is if the number of spam filters that have decided on "spam" equals (or is greater than) the min-spam value, a "spam" pre-decision can be made ($\#spam \geq min_spam \rightarrow spam$). Analogously, a "ham" pre-decision can be made if the sum of "spam" and outstanding results is less than the min-spam value ($\#spam + \#missing < min_spam \rightarrow ham$).

One disadvantage of this approach is that it does not consider "ham" results. For example if 2 of 100 spam filters vote for "spam" and the other 98 for "ham," a message is still branded as spam in total. The MiSDeM does not consider the number of "ham" results against the number of "spam" results.

Furthermore, the MiSDeM treats every spam filter equally. It does not consider whether they have calculated correct results in the past. Particularly, the number of false positives that a spam filter has been responsible for is not taken into account.

2.3.2 Related and Future Work

For the future, we plan to implement a decision maker that automatically adjusts the min-spam value in the case of too many misclassifications. Moreover, considering different weights for filters depending on their specific number of false positives appears promising.

A similar approach is taken by SpamAssassin [140]: Each filter adds its specific score to the final result. However, the scores have been determined a priori by analyzing a large corpus of spam and ham emails and are not adjusted unless they are re-trained on a new corpus.

Sakkis et al. describe two *stacked generalization* techniques, which they use to combine two machine learning filters [82]. In the SpamGuru project [88], a *super-classifier* combines the weighted results of each filter. Similar strategies have been analyzed in the EMT systems [44, 87]. A good overview is given in [49].

All these techniques indicate an advantage of combined results over individual results. However, these techniques have been applied to filters whose results reflect a confidence value, for instance a value between 0 and 1. It has to be examined how they would perform on a large set of binary results, which are returned by most Spamato filters.

2.4 Spamato Plug-Ins

Besides the plug-ins described so far and the filters, which we detail in Chapter 3, the Spamato system contains a variety of additional "productive" plug-ins. These plug-ins are not directly involved in the filter process—they have no impact on the filtering success. However, they are an added value to the user experience or indirectly support filters. In the following, we highlight only the most relevant plug-ins and show how they are integrated as pre-processors or post-processors into the Spamato framework. Components of minor importance are summarized in Section 2.4.7.

2.4.1 Local Web Server

The *local web server* plug-in, also referred to as the *web config* plug-in, provides a way to let other plug-ins manage their settings via a common web-browser interface. In fact, the local web server itself can also be configured using its own capabilities. Most filters and the other plug-ins described in this section use this facility to interact with users.

Our first approach to configuration was to provide a dialog-based Java Swing UI. But starting the work on the Spamatoxy, this attempt was rendered useless, as the proxy does not necessarily run on the user's computer but can reside on any remote machine—probably running directly on an email server. The web-browser interface closes this gap and enables Spamato to be employed in practically any environment.

2.4.2 Statistics Engine

The *statistics engine* is a post-processor which collects information about the decisions calculated in the filter process. It is also a report- and revoke-processor in order to be notified when users recall decisions of the filter process. Particularly, we are interested in the number of detected spam messages and the number of false negatives and false positives. Furthermore, the statistics engine analyzes the performance of filters, that is in how many

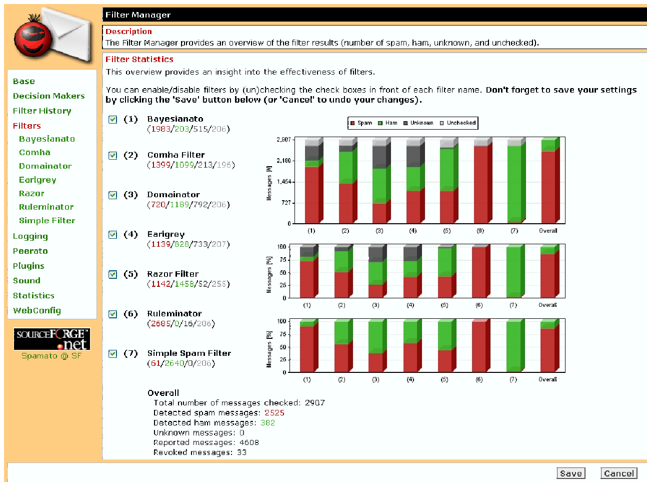

Figure 2.9: The filter statistics overview shows the number of checked emails and detected spam and ham messages for each filter.

spam detections (and false positives) each filter has been involved. The gathered data is processed locally on the user's computer and presented on the filter overview page as illustrated in Figure 2.9.

The data is also sent to a statistics server for further analysis. The server stores a complete decision log; as such, it is comparable to the filter history component storing data for all users rather than for a single one. More specifically, we store for each checked email an entry that shows which filter has voted for spam or ham[2]; report and revoke entries can be linked to checks to see why they have been classified as ham or spam in the first place. In addition, the data stored contains information about the properties of each filter and the decision maker users might have modified. Finally, we also log which add-on a user employs since some filters might depend on their capabilities.

Some of the results discussed in Chapters 7 and 5 are derived from the database containing collected data from all Spamato users. Thus, the statistics engine allows getting information about the Spamato system as a whole. Thereby, it helps find possible deficiencies but also represent a working system since we can show online usage statistics on our web page.

[2]To preserve a user's privacy, we log only those ham results which contain at least one (wrong) spam classification of a filter; we do not store data about pure ham results. Furthermore, the statistics engine is of course optional—it can easily be disabled or deleted by the user.

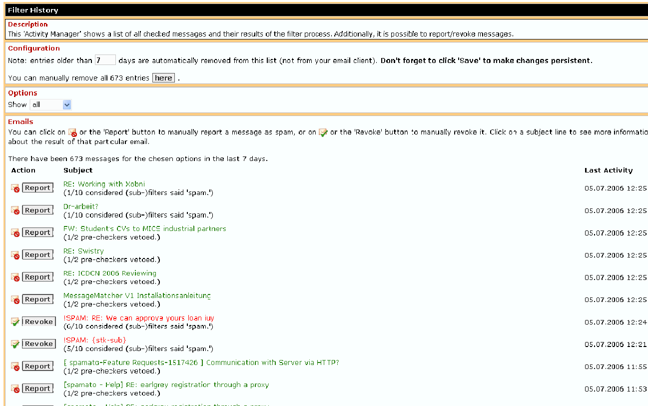

Figure 2.10: This filter history page shows an overview of filtered emails. Clicking on the subject line reveals why a message has been classified as spam or ham. With the rudimentary search function, a user can browse through particular filter results, looking for instance for emails that have been detected by a specific filter. The report and revoke buttons allow to give direct feedback to Spamato from this overview.

2.4.3 Filter History

The *filter history* is a post-processor for filter, report, and revoke events. It stores "historical" data about the filter process and logs the complete trace of all emails checked by Spamato. Users and developers similarly benefit from the filter history, as it provides useful information to both of them. Also spam filters and other plug-ins can use this data for their purposes.

Figure 2.10 shows an overview page generated by the filter history component. It presents exact details about why a message has been considered spam or ham. Furthermore, a rudimentary search function allows browsing quickly for particular results (such as spam detected by filter X) only. Users can also report and revoke emails from this overview. This feature is particularly important for the Spamatoxy in combination with POP3 accounts since in this case this is the only way to collaborate with Spamato.

The overview (as well as a more detailed view not shown here) is also helpful during the test and debug phase of the development cycle of a new filter. Looking at a web page is much easier than parsing a large log file by hand—provided that the developer prepares the data accordingly. Another part of the filter history generates charts of the processing time per filter, which facilitates to identify bottlenecks. These features make the filter history a very useful component, as it helps to detect inconsistencies even after the deployment of filters and related plug-ins.

Finally, the filter history plug-in serves as a *decision cache*, which can be queried by other plug-ins. Particularly, filters can revert to a former decision (including their own result) instead of repeatedly scrutinizing an email. For example, the Domainator (see Section 3.3) uses this cache to see whether a specific domain has been checked before. If so, and given that the check is not too old, it just returns the same result instead of searching for it on Google again—which would clearly take much longer. Thus, the caching feature indirectly increases the filtering speed and helps to consume less resources.

2.4.4 Whitelisting of Trusted Senders

The *trusted senders* plug-in is a post-processor which automatically learns trusted senders from filter decisions. A sender is identified by its email address, which is considered to be trusted if it has been seen in several legitimate emails received by the user. The actual number of necessary ham messages can be adjusted by the user with the help of the web configuration; the default value is 2.

The primary purpose of this plug-in is to serve as a pre-processor for the Ruleminator (see Section 3.4). The "Trusted Senders" rule is by default used as a ham-checker. That is, whenever a message from a trusted sender is received, the email is automatically classified as ham. Similarly, the "Distrusted Senders" rule can be used alone or in combination with other rules to mark messages as spam.

Related and Future Work

The trust values of persons are associated with the number of emails received from them. Therefore, this whitelist also reveals the most active contacts—an added value for the user since this information can be used to derive the importance of contacts like the correspondent maps in [80] or analyze social networks as in [14].

Currently, we use only the email address of a sender to identify the legitimacy of an email. Since email addresses can easily be forged by spammers, this can become a critical issue in our system. Leiba et al. describe in [56] a method to extract the IP addresses of the SMTP servers on the way from the sender to the client. This additional information could be used to distinguish between the real sender and a pretended one. Also the integration of sender verification systems, such as the Sender ID [62] or Yahoo! Domain Keys [6], could be used to improve the accuracy of our simple approach. Note however that, with very few exceptions, we have not received emails with forged known email addresses yet.

Cegłowski and Joshua Schachter implemented the Loaf [103] system to share email addresses to allow for trusted friend-of-friend relationships. They use bloom filters to abandon privacy issues: Email addresses are not shared in

plain text, but as a bit set that can only be used to query for email addresses. Loaf information could be used in our trusted senders plug-in to increase the knowledge about trusted senders not seen before—but known to a friend.

Garris et al. propose the Reliable Email (RE:) system [33] that combines the ideas of a sender verification system and Loaf with high security considerations: trusted and verified chains of friends without exposing private information. However, integrating this system with Spamato is complicated, as it relies on dedicated RE: servers.

2.4.5 Detection and Whitelisting of URLs and Domains

Three of Spamato's filters are URL-based, which means that they check whether the URLs (or rather the domains) contained in an email are associated to known spam in some way. Although parsing an email for URLs is not an expensive operation, we have outsourced this work to the *url* plug-in to avoid redundant work.

Detection of URLs

Extracting domains from a URL is sometimes a tedious undertaking. Spammers tend to obfuscate their advertised domains, which makes it hard to find the real domain. Ken Schneider gave a talk at the Spam Conference 2004 about URL obfuscating techniques [85]—most of them are still in use to confuse investigators. Also Hulten et al. report in [45] about the ongoing trend to impede the identification of URLs. We present some of these techniques and discuss our solutions.

HTML encoding schemes are rather simple problems where we can rely on the built-in Java functionality. For instance, `http://%77%77%77%2E%73%70%61%6D%6D%65%72%2E%63%6F%6D` links to the same page as `http://www.spammer.com`.

A harder task is to resolve HTTP-redirectors. For example, we can detect that `http://rds.yahoo.com/*http://www.spammer.com` actually refers to `www.spammer.com` rather than `rds.yahoo.com`.

Another frequently used trick is to add meaningless sub-domains or paths to a domain: `http://spammer@some.silly.domain.here.spammer.com.au/xyz/abc` still links to `spammer.com.au`.

Reputable hosting platforms, such as GeoCities [113], might unintentionally host spam content. For instance, let us consider the spam page on `http://geocities.yahoo.com.br/spammer`. The usual approach would lead to identifying `geocities.yahoo.com.br` as a spam domain—which it definitely is not. Our approach to this problem is to prepend the relevant path as a virtual sub-domain to the detected domain—resulting in the suspicious domain `spammer.geocities.yahoo.com.br`, which can be handled like every other domain.

Finally, URL shortening services, such as TinyURL [146], suffer from the same problem: Honest services are misused to provide spam content. For example, the URL `http://tinyurl.com/k6zdg` actually links to `www.spammer.com`. While we do not resolve the real domain, we can still handle this as in the example before. That is, we treat `k6zdg.tinyurl.com` rather than `tinyurl.com` as a spam domain.

Of course, all of these techniques can be combined by spammers and will be correctly resolved by our algorithm.

As said before, we can handle some of these problems solely with the help of Java libraries. The more elaborate techniques are tackled with an extended top level domain list, which we have derived from the Razor sources [135] as well as from [125]. This list has a simple syntax. For example, Austrian domains are described as follows:

```
(at) 2
(at).(ac|co|gv|or) 3
```

Domains ending in `.ac.at`, `.co.at`, `.gv.at`, and `.or.at` are thus considered to have 3 relevant domain parts, while all other domains ending in `.at` have only 2. For example, `www.spammer.at` resolves to `spammer.at` whereas `www.spammer.co.at` resolves to `spammer.co.at`.

GeoCities and TinyURL domains are handled similarly by introducing a second parameter which denotes how many path entries have to be considered. For example, the entry for GeoCities domains is as follows:

```
(br).(com).(yahoo).(geocities) 4 1
```

Here, the 1 denotes that the first path entry has to be prepended to a domain consisting of 4 parts as described above.

Whitelisting of URLs

The url plug-in also maintains a whitelist of known ham URLs. Domains of ham emails are automatically added to the list and gain trust in relation to the frequency in which they appear; analogously, the trust value of spam domains is decreased. Similarly to the whitelist used for trusted senders, trusted URLs can be considered ham URLs and do not have to be checked.

2.4.6 Deobfuscation of (HTML-)Text

Besides the aforementioned URL tricks, spammers also try to conceal their real intentions by obfuscating the HTML content of an email. John Graham-Cumming mentions a variety of different techniques in “The Spammers’ Compendium” [36][116]. Also Danny Goodman reports on several methods to confound spam fighters [35].

For example, writing some parts of the text in the background color confuses Bayesian-based filters. The HTML block

```
<font color='white'>T</font>s
<font color='white'>R</font>p
<font color='white'>I</font>a
<font color='white'>C</font>m
<font color='white'>K</font>!
```

will usually be rendered as ␣s␣p␣a␣m␣! rather than TsRpIaCmK!. The first combination of letters attracts the user, while the latter cannot correctly be classified by a filter for the same reason as it would not be readable by humans. One possible countermeasure is to reveal the real spammer's intention by purifying the text to appear as spam! to filters.

Many other examples exist where spammers add bogus HTML tags, replace characters with similar looking ones[3], embed <form>-tags, or play around with the capabilities of cascading style sheets. Even worse, spammers regularly develop new ideas on how to abuse HTML to hamper the filtering success.

To address this issue in Spamato, we introduced the *deobfuscator* plug-in, which calculates a purged text form of the message as a pre-processor. Filters such as the Bayesianato (see Section 3.2) can request the plain text to prevent being confused by weird tokens. Furthermore, the mere fact that such HTML obfuscation methods are employed in an email is a good indicator for spam; the Ruleminator (see Section 3.4) relies on such information in its classification round.

The deobfuscator comprises several countermeasures to help identify the actual content of an email. However, as spammers refine their attacks [92], it is only able to cope with some commonly known techniques even though several authors [55][70][97] are committed to solving this problem. Adapting to new requirements will remain an ongoing duty. Nevertheless, once implemented, it is an easy task to add such methods to the deobfuscator plug-in.

2.4.7 Other Plug-Ins

Spamato contains several other plug-ins of lesser importance and smaller scope, which we mention here for completeness. For example, the *sound* plug-in plays a short jingle whenever a spam message has been detected, the *logging* plug-in provides a common logging facility used by all plug-ins, the *mail* plug-in facilitates the parsing of emails, the *server* plug-in bundles several utility functions we use when communicating with servers, such as for sending data to the statistics server or querying the Earlgrey filter server,

[3] The author of [150] has shown that there are about $1.3 \cdot 10^{21}$ different ways of writing "Viagra" by exchanging characters with similar looking ones, for instance "\/1@9r/-\."

Figure 2.11: With the Spamato toolbar (screenshot taken from Outlook), it is possible to collaborate with the Spamato system by reporting or revoking emails. Furthermore, it shows some informative data, such as the number of detected spam messages, and allows accessing the Spamato configuration pages.

and the *chart* plug-in creates images of graphs for results presented on the configuration web pages of a plug-in.

The *tray* and *proxy* plug-ins contained in the Spamatoxy as well as the *localserver* plug-in that connects Outlook and Thunderbird with Spamato will be discussed in Section 2.5. We also integrated the *Trooth* plug-in to secure collaborative filter systems, which is the topic of Chapter 4. Finally, the plug-in system itself is partially implemented as a plug-in. We explain this interesting concept in Chapter 6.

2.5 Spamato Add-Ons

An add-on is to Spamato what an email server is to an email client: the source of all emails. An add-on connects an email client to Spamato so that incoming emails can be checked without user interaction. Emails are automatically forwarded to the Spamato system, where they are scrutinized in the filter process. Its decision is returned to the add-on, which acts on behalf of Spamato: Detected spam messages are moved to a special "junk" folder whereas legitimate emails are directed into the inbox. In short, an add-on provides the means to make Spamato a user-friendly spam filter system which is capable of transparently replacing the default spam filter solutions built into many email clients.

In case of misclassifications, the add-on addresses a second task: It allows the user to give feedback to the Spamato system by reporting or revoking emails directly from the email client. For this, it is integrated into the email client's user interface—usually by providing a Spamato toolbar as shown in Figure 2.11. The described mechanism can also be used to train (machine-)learning filters such as the Bayesianato (see Section 3.2). Furthermore, the Spamato toolbar enables a user to participate in collaborative spam filter networks, such as harnessed by the Earlgrey and Razor filters (see Sections 3.6 and 3.7 for more information).

The *Spamato Core* is written in Java and is completely independent of any email client or platform. However, the development of add-ons inherently depends on the email client and their implementation varies largely: We have to cope with different programming languages, operating systems, audiences, and generally depend on the "goodwill" of the development team to intelligently open their client for third-party integrations.

In the next section, we list some requirements which an email client has to support in order to allow the implementation of an add-on. In Sections 2.5.2 to 2.5.4, we then describe our add-ons and discuss some of their pitfalls. Finally, in Section 2.5.5, an alternative approach is presented as a proxy component which is completely independent of the email client. It is not embedded into an email client but resides between the client and an email server. Thereby, it is able to intercept and check incoming emails before they are forwarded to the client.

2.5.1 Requirements

An email client has to support the development of add-ons by providing an extension mechanism that allows changing and enriching its default behavior. This is usually achieved with an API that opens the client for foreign developers. The API gives access to the client's internal data, such as emails or folders, and allows the developer to manipulate it. Furthermore, the email client has to provide extension points, also called an event or callback mechanism, that allows performing user actions upon specific events, such as the arrival of new emails. Finally, it has to be possible to be able to modify the user-interface in order to interact with the user.

An alternative solution is to directly change the email client's source code. This is possible, though, only for open-source projects or for an "inhouse" development, where the source code is available. The final product would then be a new email client into which the Spamato system would be seamlessly embedded. However, the deployment of such a solution is difficult: Users would have to switch to a new email client, which would cost the user much more effort than the optional (and easily reversible) installation of an add-on. Therefore, we only follow the idea of adding Spamato as a standalone product rather than embedding it directly.

For Spamato, we have identified several requirements which must be met in order to enable a client to interact with Spamato. Figure 2.12 illustrates a brief overview of the necessary functionality, which has similarly been implemented in all add-ons.[4] We present the four main concepts as follows:

[4] The implementation of these concepts in the Spamatoxy described in Section 2.5.5 differs slightly, as it is not integrated into an email client. But they have been adopted and are similarly applied.

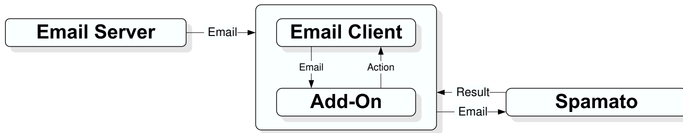

Figure 2.12: This figure illustrates the general concept of an add-on: The email client notifies the add-on of incoming emails and the add-on checks them using Spamato. The result is handled by the add-on, for instance by moving spam messages to the "junk" folder.

Launching Spamato

To make use of Spamato, it is necessary to launch it when the email client starts. An optimal solution depends on two features: First, the email client has to provide an "OnStartup" event to which we can subscribe. And second, it needs to be capable of launching external applications. Particularly, we have to start a new Java process in which the Spamato application runs. Desirable but optional is also an "OnStop" event, which can be used to reasonably shut down Spamato when the client stops.

An alternative for the first feature is to start Spamato only when it becomes necessary for the first time. That is, the launch would be delayed and executed upon another event, for instance when a new email arrives. The second requirement can also be achieved by starting Spamato independently of the email client, for example manually with its own launcher or whenever the operating system starts.

Handling Email Events

Besides the aforementioned "OnStartup" event, notification of different *email events* is also necessary. Most importantly, the add-on has to know when a new email arrives. This event should also contain information about the email itself to prevent the add-on from searching through all folders to locate it.

Furthermore, the user can manually report and revoke misclassified e-mails. Whenever the user clicks a button in the Spamato toolbar, the email client has to provide the add-on with the associated information allowing it to handle the selected emails accordingly.

Finally, the email client has to support a feature which we call a *re-check*. We perform re-checks in regular intervals, for instance hourly, to repeatedly check unread emails in the inbox that have been classified as ham so far. For this, it is necessary to be able to access all folders and emails and check whether they are read or unread. The motivation for this feature is that—in the meantime between two check attempts—collaborative spam filters might

have learned that the considered email has to be classified as spam rather than ham.[5] A re-check can thus better reflect the collected information of such filter networks. Of course, only people who do not check their email for a long time benefit from this feature, for instance when running the email client all night. Therefore, this feature is optional rather than compulsory. However, a re-check executed in very short intervals, for instance every 10 seconds, can also be used as a viable approach if the email client does not provide any "NewMail" events as described above.

Communicating with Spamato

After an email event has been sent to the add-on, it is forwarded to Spamato. To accomplish this, it is necessary to communicate with Spamato in some way. On the one hand, check, report, and revoke commands have to be sent to Spamato. On the other hand, the replies to check commands have to be received and parsed.

Communication can take place in two ways: either by using the input and output streams associated with the launched Spamato process, or by spawning a local interprocess or socket connection to a Spamato plug-in. We have chosen the second approach, as this allows to open several connections in parallel and eases the concurrent handling of requests.

Taking Actions

Once Spamato has classified an email and has sent the result back to the add-on, actions need to be taken. For example, detected spam messages have to be moved to a special "junk" folder or messages have to be marked as "checked by Spamato." In either case, it is necessary to interact with the email client, which has to execute the according actions on behalf of Spamato. Similarly, in case the user manually reports or revokes emails, the messages have to be moved to the "junk" folder or back to the inbox, respectively.

In future versions of Spamato (see Section 2.6), more elaborate actions might have to be executed. For instance, emails could be highlighted in different colors, reordered to reflect priorities, or we might even add new GUI elements to show additional information about an email. In general, the API of an email client has to be powerful enough to enable such extension ideas and to make their implementation as feasible as possible.

[5] Anecdotally, the author of [104] compares this approach in a humorous way with Schrödinger's (or rather Heisenberg's) uncertainty principle: As long as a user does not check the inbox, unread emails are in an "uncertain" state—they are both spam and not spam at the same time until observation. That is to say, a spam filter should monitor unread emails incessantly until the user reads them. This is what the re-check mechanism is supposed to do.

2.5.2 Spamato4Outlook

Spamato4Outlook (S4O) was the first add-on to be implemented. The lessons learned here have helped to develop the other add-ons. S4O is a Windows-only add-on bound to Microsoft Outlook 2002/XP and 2003. Older versions of Outlook and Outlook Express are not supported, as they are either more restrictive (Outlook 2000) or provide a completely different extension mechanism (Outlook Express).

The add-on was implemented using Visual Basic as well as Visual C# for some utility methods. The Outlook Object Model (OOM) exposes many internals of Outlook to add-on developers. It is possible to fulfill all requirements defined above; subscribing to the necessary events as well as accessing and manipulating data is possible. Also, adding the Spamato toolbar is sufficiently supported by the Outlook API.

The communication part was implemented using a socket connection to a *local server* plug-in, which is started as part of the Spamato system. We used a simple messaging scheme where all commands (check, report, revoke) were embedded into an XML-styled message. For instance, for a check, we sent the entire email (as far as it can be recovered from the OOM) from the add-on as plain XML-text to the local server and waited for a reply containing the result—either "ham" or "spam."

This approach has three advantages: First, processing XML data is supported by several utility libraries which make it easy to parse such messages in Visual Basic as well as in Java. Second, a message in the XML format can contain several entities of information (such as the request type, the source code of the email, and the spam/ham decision) and can be extended to provide for future requirements (such as to mark a message green or to set it to high priority). And finally, this solution can easily be ported to other email clients. Particularly, it is obvious that all email clients are able to open a socket connection, as email clients are inherently network-enabled. Furthermore, XML is a de-facto standard and, therefore, we can assume that XML libraries also exist for other clients. In fact, the same solution has been re-used with only slight modifications in the add-on for Thunderbird (see Section 2.5.3).

Unfortunately, using the OOM was not always satisfactory. For instance, security warnings appearing when trying to access the body of an email rendered a solution for Outlook 2000 useless. In Outlook 2003, it is still impossible to get the full header information of an email or to access the complete source text as received by the client.[6] Furthermore, we had to work around a bug in the event system that prevents Outlook from firing a "NewMail" event when more than 16 emails arrive simultaneously. In this

[6] Additional tools, such as Outlook Redemption [143], could be used to ease the implementation. However, they are neither open-source nor licensed under the GPL. Therefore, we have not been able to use such solutions in the context of Spamato.

case, we had to use the re-check mechanism to detect unread messages in the user's inbox.

Another problem we had to solve was the enhanced feature set that Outlook provides, which includes, apart from emails, also contact, to-do, and calendar entries. Furthermore, email accounts can be backed by a POP3, IMAP, Exchange, or web-mail server. While most of these features are shielded from the developer, still some details are exposed and must be handled appropriately.

Overall, implementing the add-on for Outlook has been a tedious but successful task. It works well for most people working in different environments and is the number-one add-on according to the download statistics on SourceForge.

2.5.3 Spamato4Thunderbird

Spamato4Thunderbird (S4T), like most other extensions for Thunderbird, is written in JavaScript; the XML User-interface Language (XUL) in combination with Cascading Style Sheets (CSS) is used for the user interface. The extension mechanism of Thunderbird is well engineered although it lacks a good documentation.[7]

All requirements have been met; Thunderbird provides useful events when starting up, upon email arrivals, and when shutting down. For the communication part, we have been able to adopt the local server of S4O modifying only one property: Not the actual email is sent with the XML message, but a reference to a local file in which the message has been stored by Thunderbird is transmitted. The local server can read the complete source text of the email from a file and forward it to the filter process. Having the original source text enables us to parse the complete message, access all MIME-parts, detect anomalies, and handle attachments. This is a major advantage compared to the Outlook solution, as it opens a variety of new filter possibilities.

Of course, also the extension mechanism of Thunderbird has some problems. For instance, most method calls that might take a longer time to be completed, such as downloading messages from an email server, are handled asynchronously. This makes programming more complicated, as the logic of a single activity is spread over several methods called by the Thunderbird framework rather than by the extension itself.

In contrast to S4O, S4T is platform independent; it runs on Windows as well as on Linux without further modifications. As argued above, the local server plug-in served well as a general-purpose solution for email clients other

[7] The only useful documentation can be found as part of the online reference on XULPlanet [152], which extracts its information from the Mozilla source code. Fortunately, with the increasing popularity of Firefox and Thunderbird, a large number of additional information can be found on the Internet. However, a complete official reference is still missing.

than Outlook. Although it was not possible to make an exact copy of the Outlook add-on, we took advantage of the lessons learned in the previous implementation, which significantly eased the development of S4T. Adding further requirements to support new Spamato features can now easily be implemented in both add-ons since they share many similar concepts.

2.5.4 Spamatozilla

Spamatozilla (S4M) is the Spamato extension for the old-fashioned Mozilla Mail, which comes as part of the Mozilla Suite (now known as SeaMonkey). It is very similar to S4T, but differs in one important aspect: it misses the local server plug-in. Instead, we can make use of Java directly from the JavaScript code using a facility called *LiveConnect*. LiveConnect enables us to call the Spamato filter process without using a socket connection—the JavaScript code has full access to all Java classes.[8]

The LiveConnect bridge eased the aforementioned problems regarding the asynchronous handling of method calls via callback functions. Instead, it is possible to execute a usual "`spamato.checkMail(mail)`" call and wait for its answer. However, as Spamato is a very advanced extension that accesses the network and writes to the local file system, the Java security system, by default, blocks most initiated activities. To overcome this restriction, it was necessary to modify the user's security settings before starting Spamato—an operation which is not directly supported by the email client.

2.5.5 Spamatoxy

The Spamatoxy (S4X) is special, as it is no genuine add-on. It is not bundled with a specific email client but provides a general solution for any otherwise not supported email client, including Outlook Express, Pine, and Apple Mail. Nevertheless, we call it an add-on since it performs the same task: filtering emails with the Spamato system.

The Spamatoxy is an email server proxy. That is, it resides between the email client and the real email server. All commands are transparently tunneled through the proxy before being delivered from the client to the server and vice versa. The proxy checks all incoming emails using Spamato and performs actions depending on the results. Regarding a spam message, it can either mark the message with a "SPAM" notice in the subject line, add a "Checked By Spamato" header, or directly move the message to the "junk"

[8] Notice that Spamatozilla was implemented before Thunderbird became popular. That is, Spamatozilla is the predecessor of S4T although we skip this detail in our discussion. This is also the reason why we did not re-use the complete S4T code but instead tried the LiveConnect bridge. On the other hand, we were not able to adopt the LiveConnect solution for S4T since LiveConnect relies on Sun's Java Plug-in, which is part of the Mozilla Suite but not of Thunderbird.

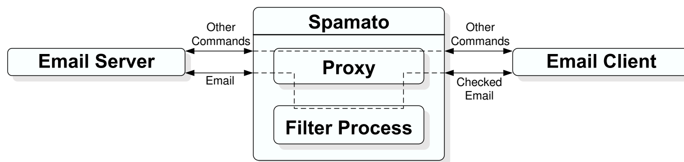

Figure 2.13: This figure illustrates the concept of the Spamatoxy. Emails from the server are intercepted, checked by Spamato, appropriately marked, and then delivered to the email client. Commands not related to the download of emails are transparently tunneled by the proxy.

folder. Messages classified as ham are forwarded without special comment. In Figure 2.13, the basic design of the proxy system is depicted.

The Spamatoxy is written entirely in Java and runs as a plug-in in the Spamato framework; as such, it can be used on all Java-enabled platforms. It is usually bundled with a tray icon component that is used for similar tasks as the Spamato toolbar. Since the handling of the tray bar varies on different platforms, it must be implemented in a way that depends on the underlying operating system. But the tray icon plug-in is not essential for the functionality of the Spamatoxy; thus, it also runs in a console-only environment.

Using the web config plug-in, it is possible to define several "services" that refer to email accounts which the proxy should take responsibility for. The proxy supports POP3 as well as IMAP accounts[9]; both types can be used with SSL encryption. In order to employ Spamato, users only have to alter their server settings in the email client, now pointing to `localserver` and the specified service port. For POP3 accounts, it is also necessary to define a new filter rule (in the email client) that moves emails whose subject line begins with "SPAM" to the "junk" folder.

One of the most difficult issues when implementing the Spamatoxy was that email clients and servers do not always adhere perfectly to the RFCs [130, 117]. Although we depend only on a very restricted subset of commands of the email protocols, different clients and servers often do not handle them in compliance with the standards. That is, if a client or server has not been tested with our proxy, it might limit the functionality of the Spamatoxy. However, fixing such bugs is usually a matter of minutes when users provide us with the relevant information.

In contrast to the other add-ons, the Spamatoxy can be run on any machine—it is not restricted to the user's local computer. It can even be run in parallel with the email server. This provides for an additional fea-

[9] Pure Web-mail providers such as Hotmail or Yahoo! Mail can be connected to using additional tools such as MrPostman [121].

ture: A single Spamatoxy installation can be used to serve several different users concurrently. However, the current version of the underlying plug-in mechanism does not allow to distinguish between different users. That is, all users would have to share a single data set, for instance for trusted users or good and bad tokens. Nevertheless, this property of the Spamatoxy is far-reaching: Once the plug-in mechanism supports multi-user installations, Spamato can also be employed in the server-side spam filtering domain that is currently not addressed by any other add-on.

In the following, we will present some details regarding the different implementations of the POP3 and the IMAP services.

Handling POP3 Accounts

The *Post Office Protocol* (*POP3*) is a simple protocol used to download emails stored on a server. For the proxy, the only relevant command is `RETR`(ieve); all other commands can be relayed without further consideration. A `RETR` is sent from the client to the server in order to download the email specified by a message number; the reply of the server contains the email's plain source text as received from the original sender.

Before the proxy delivers an email to the client, it reroutes it through Spamato. The classification result from the filter process (spam or ham) can be expressed either by flagging the subject line with a "SPAM" tag or by adding a "X-Spamato: YES/NO" header. The marked email is then forwarded to the email client, where an appropriate action can be taken by applying a filter rule (e.g., move spam to the "junk" folder).

Using POP3 accounts with the Spamatoxy has two main disadvantages. First, all emails have to be checked sequentially because the email client can request them only individually due to protocol limitations. That is, the filter process is the dominant factor determining the delivery speed since all emails have to be checked before they can be delivered to the client. It would be possible to implement a *pre-fetch* mechanism downloading and checking several emails simultaneously. But this would not necessarily reflect the user's will, wasting much bandwidth in the worst case.

The second point is concerned with the capability to provide feedback to the Spamato system—for learning and collaborative filters—and is usually achieved through the Spamato toolbar. Our first approach was to implement an SMTP feedback channel. That is, emails have to be forwarded through a special SMTP service of the proxy in order to be handled as reports or revokes by Spamato. However, emails are often forwarded inline, without headers, and enriched with automatic footer text by email clients; by default, only few clients forward email in an untouched form as an attachment. Thus, it was not possible to rebuild the email in its original form. Now, for POP3 accounts, the only convenient way to report and revoke emails is provided by the filter history plug-in (see Section 2.4.3).

Handling IMAP Accounts

The Internet Message Access Protocol (IMAP) is more elaborate than POP3, containing a variety of commands to access and manipulate emails and folders. Fortunately, as in POP3, we have to consider only one relevant command: The `FETCH` command is used by the client to order one or multiple emails from the server.[10] The server answers with the full source text of the requested emails, which can be checked by Spamato.

We have experimented with two different approaches to handle `FETCH` commands. For our first solution, we opened an additional connection to the email server used exclusively to download emails. This approach promised to decrease the complexity of parsing and reassembling commands on the main connection. However, one drawback outweighed the advantages, as we realized in our practical tests: Some IMAP servers do not allow more than one connection from the same IP address or for the same user. Consequently, the second download connection is either not established—we cannot access the emails—or the main connection is being dropped—the email client closes the connection to the proxy.

Therefore, we have chosen another solution, where we intercept all commands that are exchanged between client and server. Whenever the server now sends an email message, it is detached from the command stream and checked by Spamato. All other commands are tunneled without interruption. In case the email is classified as spam, we are now able to move it automatically to the "junk" folder on the server without notifying the client; ham messages are just forwarded to the client.

The second solution is being complicated, as different commands can be intermixed, several `FETCH` commands and their replies can be nested, and the client as well as the server are allowed to send commands to the opposite party at arbitrary times. Furthermore, some IMAP servers introduce optional or even proprietary commands which can confuse the proxy and the client. Nonetheless, our current Spamatoxy has proven to run well in different environments.

The feedback capabilities of the IMAP proxy have significantly been improved compared to the POP variant. We have introduced the concept of *intelligent folders*, which can be used directly from an email client. Whenever a user moves a message to the "junk" folder, it is considered to be spam and, thus, handled as a report in the Spamato system. Similarly, moving a message from the "junk" folder to the inbox relates to a revoke. This way, collaborating with Spamato is as easy as it is with the Spamato toolbar. Of course, using the filter history plug-in for reporting and revoking is also possible.

[10] The `FETCH` command takes either a sequential message number associated with the current ordering in a folder or, as a `UID FETCH`, a unique message number remaining fixed even after a folder content has been modified (addition or deletion of emails). We address both as `FETCH`, as they can similarly be handled.

2.6 Concluding Remarks

In this chapter, we have shown that Spamato is extendable in several ways. The framework allows adding spam filters, custom filter process and decision maker components, as well as plug-ins for many purposes. Spamato can be embedded into email clients or supplied with emails in any other way. Spamato offers a variety of extension points. It has been implemented in Java and runs on all major operating systems.

In this section, we give a brief outlook on our future plans for Spamato. Currently, the main task of Spamato, as the name promises, is to fight unsolicited emails in a user's inbox. In this thesis, we describe several techniques that help to identify messages that should rather be *outside* of than *in* this box. Although the detection of spam is one of the main concerns users are confronted with daily, it is not the only issue they face. Especially those users who receive large volumes of emails each day often feel overcharged and helpless handling the sheer amount of messages.

The term *triage* denotes "a process in which things are ranked in terms of importance or priority" (answers.com). Regarding emails, a triage system can, for example, suggest which emails should be handled first and for which an answer can be delayed for another day. It is also possible to categorize emails by topic, to provide social background information, to analyze the user's email behavior, or to collect and automatically process useful information such as appointment dates from an email. For more information on email triage see [23, 68, 12, 48].

Generally, Spamato is able to integrate email triage functionality. In fact, some of the plug-ins, such as the senders whitelist, already provide data that goes beyond mere spam filtering and could be useful to help organize emails. But two conceptual things are missing in order to employ Spamato as a full-fledged triage system: a mechanism to present triage information to the user and a mechanism to receive appropriate feedback from the user.

Our current implementation allows only to move messages within the email client and to present some information on web configuration pages. Using the Spamato toolbar, the user is also able to give minimal feedback (reporting/revoking emails) to the Spamato system. But this is not sufficient. An email triage system should seamlessly be integrated into an email client and provide further capabilities. For example, the reordering of emails or the highlighting of important messages in different colors seems to be a useful procedure to support users. Also, the embedding of additional information—such as the number of emails exchanged or the summarized content of previous topics in the same thread—directly into the email view can help users to organize their inbox. Furthermore, the user must be able to revise, annotate, and change the data provided by the triage system in order to train the system and to better reflect the user's demands.

From a technical point of view, to fulfill these requirements, it is necessary to extend the current communication protocol between Spamato and an add-on as well as the capabilities on both sides. On the one hand, Spamato will not only send filter results but also triage data to the add-on. In turn, this information has to be parsed and somehow mapped to the API and the user interface of an email client. On the other hand, the add-on has to enable users to interact with the Spamato system in an advanced way. This requires an easy to use user interface and will result in additional commands being sent to Spamato and its triage plug-ins.

Besides using the email client, add-on, and web configuration, we are also investigating other facilities to ease the interaction between users and Spamato. For instance, we have implemented a gadget for the Google Sidebar that shows information about the latest emails which have been checked by Spamato. Thereby, we are following the ideas of tools like SNARF [69] that present triage information outside an email client. Although email and triage data is then partially separated, this solution offers additional possibilities such as a cleaner user interface.

The long-term goal of the Spamato project is to turn into an "Emailato," which almost automatically handles all kinds of emails on behalf of the user. Emailato would be like a personal, smart secretary who sifts the emails and forwards only the most important ones at the right time with the necessary side information to the user. Organizing the inbox would become an easy task. *Solving* rather than *searching for* the real problems could again take most of the user's attention.

Chapter 3

Spamato Filters

This is not SPAM, it's OPT-IN MAILING.
(AlexJ <webmaster@virginsmania.com>, 5/28/2006)

The Spamato system has been designed to leverage several spam filters in order to achieve a high filter accuracy. In Chapter 2, we have describe how the filter process invokes spam filters and how their results are combined using a decision maker. As mentioned in the introduction of this dissertation, a variety of different filter techniques exist. In this chapter, we present the default spam filters which are deployed with the latest release of Spamato.

We start with summarizing the previously mentioned *pre-checkers* in Section 3.1. The *Bayesianato* is a classic Bayesian-based, statistical spam filter which is described in Section 3.2. The *Domainator* is a URL-based filter that classifies messages based on information found on search engines such as Google; it is presented in Section 3.3. Section 3.4 covers the rule-based, heuristic *Ruleminator* filter. In the last three sections of this chapter, we present different *collaborative* spam filters. We detail the general concepts of collaborative filters in Section 3.5. Thereafter, the *Earlgrey*, *Razor*, and *Comha* filters are described in Sections 3.6, 3.7, and 3.8, respectively.

We analyze different collaborative spam filters in Chapter 5. A comparison of our spam filters and real-world experiments are presented in Chapter 7.

3.1 Pre-Checkers

Pre-checkers (or ham-checkers in the terminology of the FuFiP, see Section 2.2) are special spam filters which can veto against the further processing of an email if they definitely know that an email is ham. In this case, the filter process will be aborted without using any subsequent spam filters. We make use of this concept in the following three cases.

Veto Rules (Ruleminator)

By default, the Ruleminator (see Section 3.4) contains a rule that uses the trusted senders plug-in (described in Section 2.4.4) to veto for ham when a message from a trusted sender is detected. The rule can also be combined with other constraints, or inverted to filter out distrusted senders (in which case it would not be a pre-checker anymore). Furthermore, the Ruleminator also provides the possibility to declare user-defined “veto” rules. For example, it is possible to specify particular subjects or header entries that identify known ham emails from newsgroups.

Verified Content (Trooth)

The Trooth system (see Chapter 4) uses a challenge/response mechanism to verify the email address of a user. The challenge email received by the user contains a special identifier which the Trooth pre-checker is looking for. When detected, such an email is further processed to finish the registration and can definitely be considered legitimate.

Revoke Protection

When a user revokes an email, the message is usually moved back to the user’s inbox. As new emails in the inbox are checked by Spamato, it is possible that the message is being classified as spam again. This pre-checker makes sure that revoked emails are always considered to be ham.

3.2 Bayesianato

The Bayesianato is a statistical spam filter based on Paul Graham’s popular essay “A Plan for Spam” [114]. The basic idea of this filter is to take all words contained in an email and then to combine their spam probability in accordance with Bayes’ theorem. Graham’s work has initiated numerous theoretical and practical projects in this area; we discuss some of them in Section 3.2.2. For the Bayesianato, we have taken many ideas from [98], which describes the most important criteria of statistical spam filters and is a comprehensive reference for general information on this topic.

Tokenization

In the domain of statistical spam filtering, we process *features* or *tokens* extracted from emails. The procedure of extracting these tokens is referred to as *tokenization*. There are several ways to tokenize an email. In the simplest case, tokens are just single words. Advanced tokenization techniques combine several consecutive words, try to reassemble and deobfuscate odd-looking words or phrases, and use special markers for header and URL information.

By default, the Bayesianato considers single words, no punctuation such as "!" or "?", and handles domain information as a single token such as "`www.spammer.com`". The user can configure the tokenization behavior, for instance to deobfuscate HTML code before tokenizing emails (using the deobfuscator plug-in as described in Section 2.4.6).

Calculating Token Probabilities

The spam probability of a single token can also be computed in different ways. A simple method to calculate the spam probability $P(t)$ for a token t is given as $P(t) = \frac{N_{spam}(t)}{N_{spam}(t)+N_{ham}(t)}$, where $N_{spam}(t)$ denotes the number of occurrences of token t in a spam message and $N_{ham}(t)$ the corresponding number related to ham messages. In the Bayesianato, we use the more sophisticated formula taken from [96] with the default values for the parameters set to $C_1 = 1$ and $C_2 = 2$:

$$P(t) = 0.5 + \frac{N_{spam}(t) - N_{ham}(t)}{C_1 \cdot (N_{spam}(t) + N_{ham}(t) + C_2)}$$

Calculating the spam probability with this formula has the nice property that the value will be around 0.5 for low occurrence numbers—expressing a high level of uncertainty as one would expect in case a token has rarely been seen before.

Combining Token Probabilities

The last component of a statistical classifier is concerned with the combination of the single token probabilities. Paul Graham suggests to take $n = 15$ to 20 of the most significant tokens—that is, the tokens with the highest and lowest spam probability—and calculates the overall *spamminess* S according to Bayes' theorem as:

$$S = \frac{\prod_{i=1}^{n} P(t_i)}{\prod_{i=1}^{n} P(t_i) + \prod_{i=1}^{n}(1 - P(t_i))}$$

Values calculated by this formula tend to be rather extreme—being near to 0 or almost 1. Uncertain results are missing but usually desired, as uncertainty can decrease the false positive rate (but also increases the number of false negatives).

Brian Burton, as many other researchers and developers, tweaked a variety of parameters in his SpamProbe system [17]. In SpamProbe, for example, a larger number of tokens are considered, which can also be used more than once; essentially, this puts more weight on frequent tokens. As a second method, we combined these extensions with Robinson's *geometric mean* algorithm to calculate the spamminess as described in [98]. The algorithm leads to values that are better distributed over the interval [0..1] and thereby

also express uncertainty more often than in the previous case. When the result is near 0.5, an email is classified as “unknown” rather than ham or spam.

Gary Robinson improved the geometric mean method and presented his *inverse chi-square* algorithm in 2003 [79], which has now been adopted in many statistical filter systems. For this method, all tokens with a spam probability less than 0.1 and greater than 0.9 are considered.

All three variants are part of the Bayesianato filter. The user can choose whether they should be employed separately, in which case all three algorithms provide their results to the decision maker, or combined such that the Bayesianato returns only one result for all of them.[1]

3.2.1 Starting from Scratch

Statistical filters are learning filters. They have to be trained in order to discern legitimate from spam messages accurately. Statistical filters rely on historical data seen in the past to adjudicate upon good and bad in the future. Therefore, they can only be as good as their training strategy, which includes the composition of the email corpus (received spam and ham messages) and the time of learning.

The email corpus should contain about the same amount of spam and ham messages from the same recent time period. Paul Graham denotes that a large imbalance in the number of emails could lead to a skewed token set as more tokens are known for one of the two classification types. However, Jonathan Zdziarski mentions in [98] that most filters should experience a good performance even with a ratio of 70:30. For the corpus, it is more important to account for emails from a recent time period because the users’ email behavior as well as the content of spam emails change over time: Recent emails contain tokens which cannot be assessed when the prior training has been conducted on obsolete data.

The “Train-Until-No-Errors” (TUNE) technique can be used as an initial attempt to train a statistical filter. This technique tries to learn from the given corpus unless no (or very few) misclassifications are being made. The TUNE approach can also be employed to re-train a filter once in a while to overcome the mentioned content drift of emails. For this, it would however be necessary to store all emails received in a specific time period, which can be cumbersome and expensive.

Many filters, including the Bayesianato, have an *auto-learning* mechanism which allows learning from emails just passed through the filter. Beginning with an empty set of tokens, every newly seen email can be used to train the filter—provided that the actual classification is known, either by manual

[1] Note that the current version of Spamato includes only the “Paul Graham” algorithm. Early results reported from beta-testers using all three algorithms indicate that the third approach performs slightly better than the other two.

feedback from a user or by adopting the overall decision of a multi-filter system (which can be corrected by the user in case it has been wrong). Once the filter has learned enough tokens, it can start to evaluate newly arriving emails on its own.

As mentioned above, to handle the content drift the token set of a statistical filter should constantly be readjusted. Common techniques are discussed, for example, in [96] and [98]. In the "Train-Everything" (TEFT) procedure, the overall decision is always fed to the filter no matter how it has classified the regarding email. However, doing this can lead to an overstimulation, as the data set becomes too volatile containing a very large number of insignificant tokens. The "Train-On-Error" (TOE) strategy overcomes this problem by learning only from misclassified emails, allowing thus to correct previous errors. Unfortunately, the TOE strategy recognizes a content drift only after producing false positives. In a multi-filter system such as Spamato, though, this is irrelevant as long as other filters overrule the Bayesianato's result.

The Bayesianato provides the described auto-learning techniques, which can be selected by the user. Using the mail archive maintained by the filter history plug-in, also the TUNE approach could be applied.

3.2.2 Related Work

The Bayesian-based and other statistical and machine-learning filtering techniques have been analyzed and improved by numerous researchers. The high number of research papers results from the fact that these techniques are not only used in the spam filtering domain but also for text classification in general. It is beyond the scope of this dissertation to describe all of them. Instead, we want to highlight a few interesting concepts, knowing that several others are missing.

Bayesian-based filtering in the context of spam filtering has first been described in [72] and [81]. However, Graham's "Plan for Spam" [114] is probably more frequently cited. A recent overview of different Bayesian-based algorithms is given in [65]; Zdziarski is also discussing several aspects in [98]. In [96], the author compares six different statistical filtering techniques. Another recent comparison of several algorithms is presented in [21].

Bayesian-based classifiers, which separate not only spam from ham but which are also used to sort emails into different categories, have been proposed, for instance in [77] and [73]. The POPFile spam filter system [115] is also able to assign emails to more than two (spam/ham) categories using a Bayesian-based approach.

Some work has been done to answer the question "How can Statistical Filters be Attacked?" To beat a statistical filter, a spammer has to find good words in order to conceal the real intention of a spam email. Different results—presented for example in [37, 93, 60, 61]—indicate that for this purpose it is necessary to send many emails enriched with random additional

words. Using this technique to bypass a single user's filter or that of a single company is possible but very expensive in terms of the number of emails that have to be sent. Infiltrating *all* statistical filters, is a futile endeavor.

A "solution" to this problem is presented by Aycock and Friess [11]. They describe a threat that allows for sending highly personalized spam emails. In their paper, they assume that spyware could be used to send emails, which cannot be distinguished from those normally sent, by reproducing a user's normal vocabulary or email style. Still, a spammer has to get his message to the receiver, for instance by adding a link to click on. That is, a multifaceted approach such as Spamato can detect such messages by other indicators.

A statistical filter engine is part of many existing commercial and open-source spam filter systems. In the SpamAssassin system, a Bayesian rule exists [140]. Greg Louis, the author of Bogofilter, has experimented with several variants of statistical filters [102]. The CRM114 Discriminator embodies six different classifiers; a Bayesian-based filter is complemented, for example, by a very effective Markovian-based one [106]. The DSPAM project "supports many different mathematical paradigms including Bayes, Chi-Square, Geometric, and Markovian Discrimination" [108].

3.3 Domainator

The Domainator is a *URL-based* filter that searches for domains found in an email on search engines. We use the number of listed entries as an indicator to discriminate between spam and ham domains and classify emails accordingly. The Domainator can be seen as a "meta" domain blacklist filter, as the search results often contain pages of public blacklists indexed by search engines. Although this technique can be used with many search engines, so far we have employed only Google in the Domainator. Besides providing data about domain names, Google has the advantage that it can also be used to gather further information such as the number of referring pages.

In the next section, we are motivating this idea further, showing that ham and spam domains usually produce very different results when searching for them on Google. Section 3.3.2 details the classification algorithm used in the Domainator, which we evaluate in Section 3.3.3. We discuss the algorithm in Section 3.3.4 and close presenting some related work.

3.3.1 Motivation and Preliminaries

We collected about 2000 domains from private spam and ham corpora, our statistics database (see Section 2.4.2), and different bookmark collections. The domains were classified as spam (800) or ham (1200) domains based on manual examination.

For each domain, we sent seven different queries to the Google search engine in order to determine the base criteria as input for our algorithm. The criteria and the queries are summarized in Table 3.1.

Criterion	Search Query
Domain	"<domain>"
Link	link:<domain>
Related	related:<domain>
Site	site:<domain>
Spam	<domain> spam
Blacklist	<domain> blacklist
SpamBlacklist	<domain> spam blacklist

Table 3.1: Search queries used to calculate the criteria values.

For example, the value of the *Spam* criterion was obtained for the domain "spammer.com" with the query "`spammer.com spam`", which searched for web pages that contain the string "spammer.com" as well as "spam". Notice that we were not interested in the search results themselves but only in the number of results found by Google.

In addition to these *absolute* criteria, we derived the following *relative* criteria: $Spam' = \frac{Spam}{Domain}$, $Blacklist' = \frac{Blacklist}{Domain}$, and $SpamBlacklist' = \frac{SpamBlacklist}{Domain}$. If the *Domain* value equaled 0, the relative criteria were also set to 0.

A manual examination of the collected data showed that ham domains had high values for the absolute criteria whereas high values of relative criteria are good indicators for spam domains. As an example, Figure 3.1 illustrates the results for the *Domain* and the *Spam* criteria. As can be seen, spam domains are clustered in the lower left area of the chart; ham domains are distributed over the whole area and generally tend to higher values. Note the logarithmic scaling of the axes. Also recall the fact that the value of the *Spam* criterion is always lower than the associated value of the *Domain* criterion, as the former is more restrictive.

Our observations are summarized in Table 3.2. These results were the motivation for the development of the Domainator algorithm as described in the next section.

3.3.2 The Classification Algorithm

Generally, we consider an email to be spam if at least one domain contained in the message is classified as such. A domain is analyzed in the following five phases.

Searching on Google

We determine the values for the seven criteria as described before. If all values for the criteria are 0, then the algorithm can be aborted. In this case, the classification of the inspected domain is "ham."

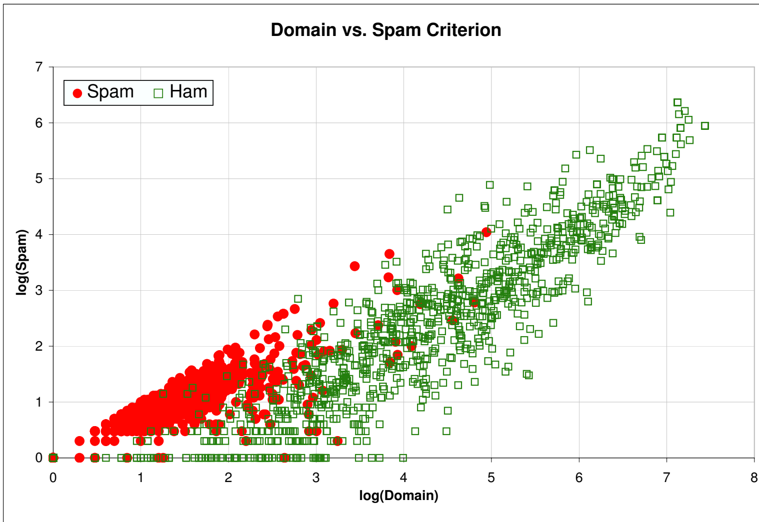

Figure 3.1: Illustration of the *Domain* and *Spam* criteria. Spam domains are clustered in the lower left area of this figure. They can be distinguished well from ham domains. Note the logarithmic scaling of both axes.

Criterion	Indicator for...	
	Spam Domains	**Ham Domains**
Domain	O	-
Link	-	O
Related	-	+
Site	O	++
Spam'	+	O
Blacklist'	+	O
SpamBlacklist'	++	O

Table 3.2: Capability of different criteria to discriminate between spam and ham domains. A criterion marked with "-" has a very low impact, "++" denotes a very high discrimination factor, and "O" and "+" are in between. Also see Table 3.4.

Calculation of Scores

We assign a positive spam score $s_i(x)$ to a relative criterion C_i if its value x exceeds a spam threshold value ST_i (see Formula 3.1); similarly, an absolute criterion receives a spam score if its value is below the spam threshold (Formula 3.2):

$$s_i(x) = \begin{cases} \frac{S_{\leq x, \geq ST_i}}{S_{\geq ST_i}} & \text{if } x \geq ST_i \\ 0 & \text{otherwise} \end{cases} \qquad \text{(relative criterion)} \qquad (3.1)$$

$$s_i(x) = \begin{cases} \frac{S_{\geq x, \leq ST_i}}{S_{\leq ST_i}} & \text{if } x \leq ST_i \\ 0 & \text{otherwise} \end{cases} \qquad \text{(absolute criterion)} \qquad (3.2)$$

$S_{\leq x, \geq ST_i}$ denotes the number of all initially known spam domains (see Section 3.3.3) whose criterion values are less than or equal to x and greater than or equal to ST_i. The other variables are defined accordingly.

Similarly, a relative (absolute) criterion C_i has a positive ham score $h_i(x)$ in case its value is less (greater) than or equal to a ham threshold HT_i, as shown in Formulas 3.3 and 3.4, respectively:

$$h_i(x) = \begin{cases} \frac{H_{\geq x, \leq HT_i}}{H_{\leq HT_i}} & \text{if } x \leq HT_i \\ 0 & \text{otherwise} \end{cases} \qquad \text{(relative criterion)} \qquad (3.3)$$

$$h_i(x) = \begin{cases} \frac{H_{\leq x, \geq HT_i}}{H_{\geq HT_i}} & \text{if } x \geq HT_i \\ 0 & \text{otherwise} \end{cases} \qquad \text{(absolute criterion)} \qquad (3.4)$$

The actual values for ST_i and HT_i are heuristically derived from the initially known "master" domain set described in the previous section. Their values are summarized in Table 3.3.

Criterion C_i	**Type**	ST_i	HT_i
Domain	absolute	9	8
Link	absolute	3	2
Related	absolute	1	0
Site	absolute	65	64
Spam'	relative	0.059	0.109
Blacklist'	relative	0.007	0.017
SpamBlacklist'	relative	0.007	0.015

Table 3.3: Thresholds for applying spam and ham scores to search criteria.

Criterion	w_i^{spam}	w_i^{ham}
Domain	8	4
Link	4	20
Related	4	80
Site	8	100
Spam'	60	10
Blacklist'	100	10
SpamBlacklist'	150	10

Table 3.4: Criteria-specific weights to express different significance levels.

Criteria-specific Weighting

In this step, we calculate the weighted and normalized sum of all spam (S) and ham (H) scores, respectively:

$$S = \frac{\sum_i (s_i \cdot w_i^{spam})}{\sum_i w_i^{spam}} \qquad \text{and} \qquad H = \frac{\sum_i (h_i \cdot w_i^{ham})}{\sum_i w_i^{ham}}$$

The weights w_i^{spam} and w_i^{ham} are criteria specific. They represent discrimination factors which reflect the different significance values of the criteria. The default values are shown in Table 3.4. They have been derived from the initial domain set and the results presented in Table 3.2.

Considering Aggressiveness

The Domainator provides a further, user-specific parameter which can be used to adjust the *aggressiveness* of the algorithm. The minimal value of 0 effectively disables the Domainator, as all domains would be considered ham; using the maximal value of 1 would classify every domain as spam.[2] The default value is $A = 0.5$.

$$S' = S \cdot \begin{cases} \frac{A}{1-A} & \text{if } A \leq 0.5 \\ 1 & \text{otherwise} \end{cases} \qquad \text{and} \qquad H' = H \cdot \begin{cases} \frac{1-A}{A} & \text{if } A \geq 0.5 \\ 1 & \text{otherwise} \end{cases}$$

Finding a Decision

The final result $R = S' - H'$ denotes whether the considered domain is classified as spam ($R > 0$) or ham ($R \leq 0$). In Table 3.5, we present example results for three manually chosen domains.

[2] To avoid a high number of false positives, we multiply the *aggressiveness* by 0.9 before calculating S' and H'.

	ethz.ch	memoryics.info	computermadnessone.com
Domain	20300000	3	54
Link	16700	0	0
Related	0	0	0
Site	14900000	0	0
Spam	128000	1	46
Blacklist	17200	1	20
SpamBlacklist	11000	1	19
Spam'	0.0063	0.3333	0.8519
Blacklist'	0.0008	0.3333	0.3704
SpamBlacklist'	0.0005	0.3333	0.3519
Score	-0.5186	0.5736	0.6464
Classification	Ham	Spam	Spam

Table 3.5: Example results for three selected domains.

3.3.3 Evaluation

We evaluated our initial domain set with the presented algorithm. Figure 3.2 illustrates the ratio of false positives and false negatives in dependence of the value of the aggressiveness. With the default aggressiveness of 0.5, there have been 10 false positives (0.8%) and 92 false negatives (11.5%). In general, an aggressiveness value between 0.1 and 0.8 seems appropriate, smaller or larger values lead to high misclassification rates.

Of course, the algorithm was tuned to calculate good results for the initial domain set. To counter this criticism, we analyzed an additional domain set that did not contain any domains of the initial set. In total, there were about 2000 domains—1000 spam and 1000 ham domains—which had been collected during the normal usage of Spamato by several participants. The domains were immediately analyzed when they arrived in the users' inboxes. As expected, the results are slightly worse than for the initial set but still acceptable: The Domainator produced 11 false positives (1.1%) and 181 false negatives (18.1%). In Chapter 7, we present further results from all Spamato users which, unfortunately, perform worse.

3.3.4 Discussion

The Domainator leverages search engines to identify spam. One problem with this method is that search engines have to index web pages before the Domainator can make use of them. While some of our query results can be obtained from the considered domain itself, other information, such as for the *Spam* criterion, need to be published by people or systems sharing their knowledge on the Internet. In other words, the Domainator can only be as good as the information provided by others. Moreover, the Domainator can

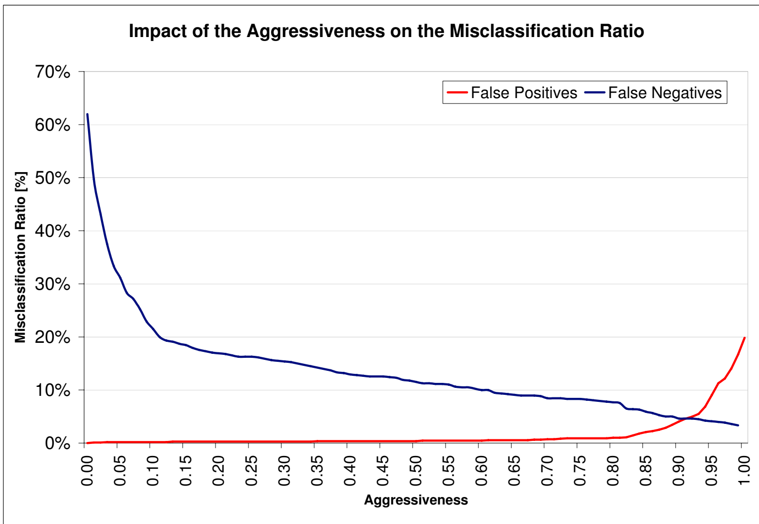

Figure 3.2: Impact of the aggressiveness value on the misclassification rate.

act only in a time-shifted way, as it has to wait for other people to publish and for the search engine to index the relevant data.

If a search engine does not provide enough information about a domain, the Domainator is not able to classify it appropriately. Treating an unknown domain—that is, one that has not been indexed by a search engine—as spam seems a bad idea, as for instance many private homepages might not be listed on a search engine. Nevertheless, the question is how often a person receives such an unknown domain. If ever, there is a high probability that it has been received from a *trusted sender* (see Section 2.4.4).

As the Domainator utilizes external information, this raises the question whether this information can be trusted. We believe that the reputation of large search engines, such as Google, is good enough to completely rely on their data. Although they might control the publication of their search index at will, they definitely do not have any interest in impeding the identification of spam domains. The question remains if people can manipulate a search engine's index to mislead the Domainator. The main motivation of spammers would obviously be to give their spam domains a reputable touch. But also the other case—lowering the trustworthiness of ham domains—seems attractive as it can lead to a high number of false positives which would question the entire filter approach. Both problems are related to "web spam" in general: Spammers try to manipulate the index of a search engine by controlling

huge server farms and thousands of domains which reference each other in order to gain a high page rank [39, 94]. Detecting web spam attacks and reducing their impact on search engine results will similarly lower the capability of spammers to deceive the Domainator. As for now, we think that this problem is neglectable and that we can fully trust the search results found on Google.

A problem with the search results is that some domains seem to be spam related although they belong to the opposite. For instance, the domains `spamato.net` and `spamassassin.org` both contain the word "spam" which results in a high value for the *Spam* criterion. In fact, the Domainator classifies `spamato.net` erroneously as spam (with the default aggressiveness of 0.5). The same problem also holds for companies which try to sell anti-spam products or work in this area. The best way to handle such domains is to rely on a domain whitelist as described in Section 2.4.5. Another problem is that some domain names might actually be contained in others. For instance, the domain `spammers.com` is contained in `anti-spammers.com`. Therefore, results for `spammers.com` benefit from the good reputation of `anti-spammers.com`. Similarly, the Domainator can treat ham domains worse than appropriate if they are part of the name of a spam domain.

Our algorithm depends on three parameters: the threshold values, the weights, and the aggressiveness. As described above, the aggressiveness is a value that can be adjusted by the user whereas the values for the thresholds and weights have been heuristically determined after examining our initial domain set. Although the Domainator performs well with the current settings, the values might be subject to change if the properties of ham and spam domains evolve. Currently, the values are fixed and therefore cannot reflect any changes. It is possible to modify the values either with each new version of the Domainator (if necessary) or leave it to the user to manually adjust them. A better solution, though, would be to automatically optimize the values based on the experience gained during the normal usage of Spamato.

Finally, we do not want to conceal the fact that the Domainator produces a high amount of network traffic. For each query sent to Google, the Domainator receives about 10 to 20 kB of data. On total, this results in 70 to 140 kB of data (per domain!) which stresses the user's network connection in addition to the inspected spam message. For the same reason, the Domainator is rather slow in deciding on spam or ham, as it has to wait for all replies. To tackle this problem, we have implemented a caching strategy to query Google only if it has not been done in a specific time interval before. Also, the domain whitelist helps to decrease the number of queries sent to Google.

Overall, we think that the Domainator is a good complement to other URL-based filters.

3.4 Ruleminator

The Ruleminator is a *rule-based* or *heuristic* filter. It allows defining static rules that classify a message based on properties of the headers or the body part of an email. For example, the word "viagra" in the body of an email is a good indicator for spam and emails that claim to have been sent years ago (or in the future) can be deemed spam as well. A rule can define a matching of different kinds: For example, they can match an exact phrase (subject *equals* "Hi"), start or end with a phrase (subject *starts with* "!SPAM"), contain a phrase (the body *contains* "viagra"), or match an arbitrary regular expression (body *matches* "(?is).*img.*?src.*?cid:.*"). In addition to spam rules, it is also possible to build ham and veto rules. Ham rules just decide on ham rather than on spam, whereas veto rules are invoked in the pre-check phase of the filter process (see Section 2.2) and can thus veto for a ham classification (without considering any spam filter).

Filter rules of these kinds can be defined in most email clients, such as Outlook or Thunderbird, without the assistance of Spamato. What makes the Ruleminator special is its capability to incorporate the results of other plug-ins and filters. For instance, a default veto rule of the Ruleminator is to classify a message as ham if it has been sent by a trusted sender—which has been verified by the trusted senders plug-in described in Section 2.4.4, not by the Ruleminator.

Rules can also be combined in order to define more powerful expressions. For instance, an email that contains embedded images and has been sent by a distrusted person is more suspicious than each of the single constraint expresses. Such a "meta" rule also allows combining the results of different spam filters, which renders the Ruleminator a kind of "pre" decision maker.

One weakness of rule-based filters is that rules are usually very specific; they identify only a particular type of spam (or ham) emails. For instance, a spam rule that looks for "viagra" in an email cannot filter emails that contain "vi@gr@" or even "cialis." The Ruleminator is able to express more powerful rules as described above. But, in general, rule-based filters tend to contain a lot of weak rules that are hard to understand and maintain. This often leads to a high number of misclassifications, which is of course undesired.

Another drawback of heuristic filters is that rules are usually static and cannot adapt to feedback from the user. For instance, the "viagra" rule will always incriminate emails that contain this keyword even if a user works for Pfizer—it cannot learn. In Spamato, some rules, such as the (dis-)trusted senders rule, rely on the capability of plug-ins which allow them to adjust their behavior and thus overcome this weakness. However, most user-defined rules are static which can again lead to a high number of misclassifications. Only a sophisticated decision maker which weighs the rules in accordance to their past performance can help alleviate this problem.

From a technical point of view, new rule types can easily be added to enhance the capabilities of the Ruleminator. Currently, we are working on an implementation to use scripting languages, such as Ruby, to define new rules without modifying the Java source code of the Ruleminator. This will allow to create very sophisticated rules at runtime—at least for users with programming skills.

Default Rules

The Ruleminator contains by default three spam and one veto rules. The "Trusted Senders" rule vetoes against further processing of an email if its sender is whitelisted (checked with the *trusted senders* plug-in, see Section 2.4.4). Similarly, the "Distrusted Senders" rule votes for spam if the sender of an email is not on the whitelist. In addition, the "Distrusted CID" rule also checks whether an email contains an embedded image (detected if a "`cid:`" reference exists). Finally, the "PreChecked" rule tests whether an email is already flagged as spam by other spam filters, for example by SpamAssassin.

3.4.1 Related Work

SpamAssassin [140] is a popular server-side spam filter which incorporates several hundreds of *light-weight* spam rules. It is usually invoked by other mail processing tools such as Procmail [132] or MIMEDefang [120]. From a technical perspective, it is similar to the Spamato system. Since version 3.0 of SpamAssassin, it is possible to extend its capabilities with plug-ins which are comparable to Spamato's plug-ins. SpamAssassin also allows to declare *meta rules* which are rules that combine the results of several other rules.

As mentioned before, most email clients provide user-defined filter rules. With such rules, most often specific emails that share a common identifier, for instance newsletters, are moved to a special folder. In other words, filter rules are used to sort known ham messages rather than to distinguish between ham and spam.

RIPPER [19] automatically learns rules to sort texts into categories. In the ifile [77] filtering system, a Bayesian approach is taken to classify emails. The FogBugz system [119] also uses a Bayesian-based filter to sort emails in a bug tracking system; this concerns the separation of "technical questions from sales-related questions" and has also been employed to filter spam emails. Pantel and Lin [73] describe the SpamCop system that is used for spam classification as well as to organize emails in general. The POPFile spam filter [115] application can also be used to detect spam besides categories for legitimate emails.

A few systems allow sharing rules among users. Jung and Jo [47] describe a system in which collaborative software agents exchange rules. The rules

are automatically generated containing email *features*, for instance embedded URLs. Garg et al. describe a similar system in [32]. Both systems, though, do not seem to be employed in a real-world scenario. In contrast, the "Camel's Eye" spam filter proxy [153] also includes a sharing facility for rules and can be used for practical purposes.

3.5 Collaborative Spam Filters

Collaborative spam filters[3] harness a collectively maintained knowledge base containing information about spam and ham emails. They take advantage of the fact that spam is sent by the millions and that users who receive such emails can help each other to sort them out. In a nutshell, the general procedure is as follows: When an email is received, it is first queried against a global database. If it is listed, the email is considered to be spam and automatically removed from the user's inbox. Otherwise, the user has to classify it manually either as ham or spam. In case of spam, the email is reported to the database and can, in turn, help to identify similar spam received by other users. Thus, the chief principle of collaborative spam filters is to share the common knowledge about spam emails among participants.

Of course, no user would like to see all emails being sent in a plain text format to a central database since legitimate emails are inherently of private nature. To overcome this problem, emails are represented as *fingerprints*—also called digests, hashes, or signatures—that cannot be used to deduce the actual content of emails; thus, the privacy of users is guaranteed. Since spammers often mischievously modify the content of an email, the most important property of a fingerprinting algorithm is to be insensitive to such modifications. Slightly modified emails have to be mapped to the same fingerprint in order to be recognized as similar in the database. That is, a good fingerprint algorithm must be tolerant to "noise" but must still be able to preserve a high discriminative factor, preventing thus false positives.

Calculating a cryptographic hash, for instance using the SHA1 algorithm [138], for an entire email text is not very promising, as even the addition of a single character to the email will result in a completely different fingerprint. Also, relying on irrelevant information, for instance considering only the sending date, will not meet the requirements. Therefore, meaningful *features* of an email are usually selected in order to achieve a stable fingerprint. Examples of common features include letter frequencies [66], significant words, shingles or token sets [47, 66, 16, 18, 52, 20], selected n-grams [99, 24, 84, 63, 31][135, 123], phrases, sentences, or paragraphs [30, 15], and URLs [47, 95, 3][135].

[3]Note that collaborative spam filtering is not *collaborative filtering*. The latter denotes a mechanism to recommend items to users of similar interest. See Chapter 4 for more information.

Note that not all of these algorithms are specific for the detection of spam. Detecting similarities in data is important in a variety of domains, including the detection of copyright infringements and plagiarism [89][107], the comparison of image and audio data [40, 58], and the optimization of storage and caching strategies in distributed file systems [76]. However, the spam domain is probably the only area in which adversaries, namely spammers, actively try to penetrate such algorithms in order to generate different fingerprints of similar emails.

An obvious weakness with collaborative spam filters in general is the *bootstrapping* problem: A few users will always experience undetected spam messages, because someone has to be the first receiving and reporting a new kind of spam email. Users who check their inbox only once a day are more likely to benefit from such filters than users who constantly check for new emails. That is, collaborative spam filters—taken on their own—are not able to guarantee a 100% protection of spam. In a spam filter system like Spamato, however, this is not a problem since other types of spam filters can bridge this gap.

A related point is that collaborative spam filters depend on the number of participating users or rather the number of reported spam emails. They become more effective as more users join their network. Furthermore, they also suffer from a *free-riding* problem where users only query the filter network without contributing to it (by reporting spam emails). In order to increase the number of known spam emails, *spam traps*, also known as tar pits or honey pots [75], can be employed. Spam traps consist of email addresses which are published on the Internet and are automatically harvested by spammers using email crawlers. As these email addresses are never used for normal communication, incoming emails can, by default, be considered spam and thus reported to the spam filter network. Nowadays, millions of such fake email addresses are part of spammers' email lists and assist the work of collaborative spam filters.

So far, we have assumed that reported emails can automatically be turned into indicators of spam. Unfortunately, the real world is more complicated. Users can, by accident or intendedly, report arbitrary emails as spam—even legitimate newsletters from Amazon or security warnings from Microsoft. Manually maintaining a server-side whitelist for known ham fingerprints can help alleviate this problem but it also increases the management effort. In order to remove such messages by the community, once a false positive has been produced, messages can be *revoked* from the system. However, a straightforward implementation of this mechanism would enable spammers to revoke all their advertisements and, thus, would render collaborative spam filters useless.

A trust or recommendation system is usually employed to observe a collaborative spam filter network. Here, reports and revokes and the identifi-

cation of their associated users are registered in a database. Users receive a *trust value* that expresses their conformity with the overall decision, for example a decision by the majority. Querying the database for the fingerprint of an email then considers the ratio between reports and revokes weighted by the trust values of the voting users. We further detail this topic in Chapter 4 when we describe the Trooth trust system, which is a plug-in in the Spamato spam filter system.

A general problem of centralized systems is their vulnerability to (distributed) denial of service (DDOS) attacks. A single server or database can easily be turned off by addressing it with continuous requests in high volumes. Blue Frog, a former collaborative anti-spam solution that effectively tried to perform DDOS attacks on spammers—which is also known as a "filters fight back" approach—, has recently been put out of business after itself has been the victim of such an attack [101]. The Okopipi project [126] aims to become a distributed, peer-to-peer replacement for the Blue Frog anti-spam solution being thus immune to this vulnerability.

The authors of [99] also describe a collaborative peer-to-peer filter that stores the collectively known fingerprints in a distributed hash table rather than in a central database. [13] describes a mechanism to efficiently search for similar files in a peer-to-peer system. [25] is an email-server-based collaborative spam filter system in which information about known spam messages is exchanged between SMTP servers. Since the SMTP network is likewise distributed, this approach is also resilient to DDOS attacks. [38] and [53] also take advantage of the SMTP but are employed on the client-side: Fingerprints (and other information) of known spam messages are exchanged using the normal email distribution channel to collaboratively fight spam.

Just like the Domainator, collaborative spam filters have to send data over the network in order to classify an email. The amount of data transferred is, compared to the Domainator, relatively small. Still, each email causes further network traffic, which might be discouraged. Due to the network latency, this filter technique also delays the evaluation of an email. To limit the data sent to a server and to decrease the time it takes to classify an email, a common approach is to use local caching and whitelisting mechanisms that store data about known fingerprint classifications. Before querying the global database, the locally available information is searched and, if a matching fingerprint is found, the previous results can be applied.

3.5.1 Related Work

We have already cited many of the related projects in this area. As mentioned, only a few of them have been employed to filter spam. Most of the work is used "offline" to analyze a given set of files for finding similar ones. A good introduction can be found in [15] and [30]. Here, we will only detail a few techniques that have particularly been designed for the spam domain.

Kołcz et al. [52] describe an improvement of the original I-Match algorithm [18] outperforming it in a spam filtering application. The basic idea of the original I-Match algorithm is rather simple: All unique words of an email that are also contained in a pre-compiled lexicon are hashed to generate a fingerprint. While this method is insensitive to changes in the order of words, the addition or deletion of words, which also have to be in the lexicon, results in a modified fingerprint. [52] proposes the usage of several lexica that are derived from the original lexicon by eliminating a small random fraction of original terms. Instead of using a single fingerprint, a fingerprint is calculated for each lexicon and one or a small number of fingerprints have to match in order to consider messages similar.

We see one major problem with this technique: The rather large lexica have to be deployed to all participating clients. Furthermore, they cannot easily be updated in order to adapt to changes in the commonly used (spam) words. Nevertheless, it would be interesting to see how the improved I-Match algorithm would perform using different lexica derived locally, for instance from each user's token set managed by the Bayesianato.

Metzger et al. [66] propose a twofold filtering technique. They call the first one a Support Vector Machine (SVM) approach, which is essentially similar to I-Match. In the second method, they determine the frequency of representative letters of each email and use the difference of the frequencies for each letter as a similarity metric: Similar (spam) emails should have about the same "letter histogram" whereas different emails do not. We have analyzed this idea and present the results in Chapter 5.

The Usenet Binary Spam Filter (UBSF) [148] is used to block unsolicited binary content (such as images) from binary Usenet groups.[4] The entire data is hashed using an MD5 algorithm. The resulting fingerprint can be queried using a HTTP request such as `http://archive.xusenet.com/ubsf.html?q=439cd6b58e60b328c3b1e4cc95be69cb`. Although the UBSF is currently not employed for email spam, we comment on it here, as images embedded in spam emails become a growing annoyance [131]. The weakness of this algorithm, however, is that MD5, as a cryptographic hash algorithm, is very sensitive to changes in the data. Since spammers have started to send slightly modified pictures, for instance by changing the background color or adding randomly distributed pixels to the picture, this technique cannot be used in this form for detecting similar emails.

Damiani et al. [24] describe a fingerprinting algorithm that is similar to Nilsimsa [123]. Nilsimsa works roughly as follows: A sliding window of size five is moved over the entire email text. For each window, several trigrams are calculated. Each trigram is hashed to one of 256 buckets and its bucket counter is increased by one. In the final 256-bit fingerprint, a bit is set to 1 if

[4] Anecdotally, binary Usenet groups are mostly used to share adult content which is also an area in which spammers advertise the most. So readers of such newsgroups might not really be averse to see spam message of this kind.

the associated bucket counter exceeds a given threshold. The threshold value is the average bucket size for each bucket previously determined by analyzing a large number of emails. Two fingerprints are considered to be similar if at least 152 bits at the same position have the same value.

Damiani et al. propose two simple modifications to this approach. First, they use the median instead of the average value when calculating the fingerprint. And second, the number of bits that have to be equal in order to consider fingerprints similar is increased to 182. In Chapter 5, we present some results for both approaches.

3.6 Earlgrey

The Earlgrey[5] filter is a URL-based collaborative spam filter. It uses the *url* plug-in (see Section 2.4.5) to extract the URLs or rather the unique domains contained in an email. It then checks whether the domains appear in the central Earlgrey database and if so, it considers the email as spam. A more detailed description of the filter process is as follows:

(1) Extract all unique domains D from an email using the *url* plug-in. If all domains in D are whitelisted, classify the message as ham and stop.

(2) Otherwise, send D to the Earlgrey server.

(3) The Earlgrey server removes all server-side whitelisted domains from D, which results in D'.

(4) A single MD5 hash is calculated of all domains in D': $H = \mathrm{MD5}(D')$.

(5) H is queried against the Earlgrey database, which contains the number of reports and revokes for H.

(6) The result $R = \frac{\#reports}{\#reports + \#revokes}$ is sent to the client.

(7) If R is larger than a user-defined threshold, the message is considered spam, otherwise it is ham.

This sequence is not quite complete because the Earlgrey filter additionally depends on the Trooth trust system in order to prevent malicious users from harming the system. Figure 3.3 depicts the complete system and illustrates the cooperation of the Earlgrey and the Trooth system. In Chapter 4, we describe the Trooth system and give a detailed example for filters using it in Section 4.5. Here, we only sketch the real process.

[5] From Wikipedia.com: “The term *URL* is typically pronounced as either a spelled-out initialism ('yoo arr ell') or as an acronym (**earl** or ural as in the Ural Mountains).” Moreover, Earlgrey is a black tea which the author of this dissertation likes to drink in the evenings. This filter has nothing in common with Greylisting [57].

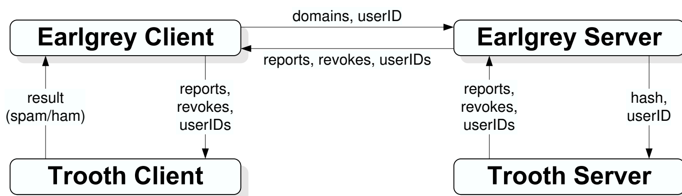

Figure 3.3: Illustration of the Earlgrey filter. Domains contained in an email and the user identifier are sent to the Earlgrey server. A hash is calculated for which the Trooth server returns the reports and revokes including their users. This information is forwarded back to the Earlgrey client. The client part of the Trooth system determines whether the original email is spam or ham.

Steps (5) to (7) are only executed as described above if R is near to one; this is referred to as the *majority heuristic* in the Trooth system. Otherwise, a selected number of reports and revokes as well as the identifiers of the users who cast them is sent to the client. The client weights the reports and revokes in accordance with the trust value of the users which the Trooth system maintains on the client side. The fraction R' of the *trusted* report and revoke values is calculated as in step (6). Finally, step (7) can be applied using R' instead of R.

Reporting and revoking of emails is performed similarly: All domains are sent to the Earlgrey server, which stores a hash of the set of all non-whitelisted domains and the user identification (verified by Trooth) in the database. To adjust the trust values of users, the Earlgrey server (in combination with the Trooth system) also returns the identification of assenting and dissenting users.

Caching and Whitelists

On the client side, the Earlgrey filter employs the caching facility provided by the *filter history* plug-in (see Section 2.4.3) to reduce the number of requests sent to the server. Before sending a request for a new message, the client searches for results in the cache and applies a previously determined result if found. In order not to conflict with *re-checks* (see Section 2.5.1), the time a result stays in the cache is shorter than the interval for re-checks.

The client-side domain whitelist is maintained by the *url* plug-in. The server-side whitelist can manually be updated by the administrator. Since client requests are logged, it is also possible to automatically promote domains that have often been queried but never reported.

General Problem: No Domains

Obviously, URL-based filters in general cannot evaluate emails that contain no URL. The Earlgrey filter classifies such emails as "unknown" rather than ham. We have analyzed 90000 emails from our statistics database: about 78000 contained at least one URL. Therefore, a URL-based filter can only evaluate approximately 86% of all emails. In other words, this means that they have a minimal false negative rate of 14% (regarding all emails).

Future Work

For the future, we plan to implement an Earlgrey extension for the Firefox browser. The extension can warn users when they open the pages of a spam domain that is listed in the database. This would also be a protection for unaware users that are directed to a phishing page.

3.6.1 Related Work

The Earlgrey filter differs from the Domainator, as it does not query a search engine but our own database. Furthermore, the Trooth trust system makes sure that no malicious user can tamper with the system.

Yeh and Lin [95] describe a URL-based technique which works with a local database rather than a global, collectively maintained one. The database can however be shared but access to it is restricted to a single company. This solution can only be successful if the user base of the system is large enough.

The Whiplash algorithm in the Razor system considers a message to be spam if one domain is tagged as spam. In contrast, the Earlgrey filter considers a message to be spam if the hash of all domains contained in an email is spam. Many emails contain only a single domain; in this case, the Whiplash and the Earlgrey algorithm calculate the same result, as there is no difference in evaluating a single domain or the hash of a single domain.

However, there is an increasing number of emails linking to more than one domain; we refer to these as *multi-domain* emails in contrast to *single-domain* emails, which contain only one domain. In [3], we investigated 13750 spam messages and discovered that about 5800 (42.4%) of them contained more than one domain and about 1000 (7.3%) emails referenced even ten or more distinct domains. The reason for this enrichment of URLs is that, for example, spammers use images in their messages that are loaded from different legal online shops or that they link to trustworthy sources to affirm their legitimacy. It is also a common practice to insert several *fake* domains to spam messages for the only purpose of misleading URL-based filters.

The drawback of the Whiplash filter is that when spam messages are reported to the Razor network, also ham domains which are probably contained in a multi-domain email are discredited. This means that for example a message which contains only a single ham domain that was reported as

part of another multi-domain email before will subsequently be classified as spam, too.

The Earlgrey filter is immune to this problem. The fingerprint of a multi-domain email, which might also contain one or more ham domains, does not conflict with any other fingerprints derived from the same ham domains. That is, a message containing the ham domain A has a different fingerprint than a message that contains A and the real spammer's domain B. However, the Earlgrey filter has another problem: Just like spam messages interspersed with random text chunks paralyze a hash-based spam filter (like the Ephemeral algorithm in the Razor filter), random insertion of constantly changing fake domains alters the fingerprint and makes it impossible for this filter to uniquely identify the message. In Chapter 7, we show how the Earlgrey and Whiplash algorithms compete against each other.

URL blacklists, such as [144] and [149], are similar to the Earlgrey filter since they blacklist domains contained in spam emails. Querying for a domain is performed using a DNS lookup. For instance, if the domain `spammer.com.sc.surbl.org` can be resolved, `spammer.com` is blacklisted by SURBL, otherwise not. In contrast to the Earlgrey filter, these blacklists provide a rather harsh "this is spam/ham" policy: There is no confidence value as it can be derived from the number of reports and revokes. Moreover, an often stated criticism of blacklists is that they are too aggressive in listing domains. It is rather hard to convince the administration of such blacklists to remove a legitimate domain once it has been added to their index. In our system, Trooth maintains trust values for reporters and revokers on the client side in order to allow for a user specific rating of domains.

DNS blacklists, such as those maintained by Spamhaus [141] and SORBS [139], do not blacklist domains contained in an email but IP addresses of SMTP servers from which spam has been sent in the past. This mechanism is especially useful, as it can be used to block emails before they are delivered to the receiver. Jung and Sit report in [46] that the MIT Artificial Intelligence Laboratory was able to block about 80% of all spam emails using DNS blacklists in 2004. However, such systems suffer from the same problem as URL blacklists.

Han et al. propose a collaborative URL spam filter to block link spam in blog systems [41]. Motivated by the work in [53] and [24], they perform an *adaptive percolation search* to find link spam among blogs which is based on random walks and probabilistic broadcasts in the graph of trusted blogs. Unlike our approach, they do not query a central database but use locally cached information or a periodic search mechanism to identify similar spam links. If the number of positive replies from other blogs exceeds a given threshold, a blog entry is considered spam.

3.7 Razor

Vipul's Razor [135] is an open-source[6] spam filter originally written in Perl. We have re-implemented a fully compliant Java derivative from the original Perl source code as well as from the protocol specification described in [142]. Our Razor implementation is part of the Spamato system and has been published under the GPL. It has also been adopted in the Camel's Eye filter written by Jörg Zieren [153].

Finding Servers

The classification of a message as spam or ham is a process of several steps in the Razor network. The Razor client regularly has to *discover* the available servers to be able to check, report, and revoke messages. The following steps are performed by the client to build a list of available servers:

(1) A set of *discovery servers* (D) is collected using the Domain Name System: The client tries to resolve the IP address of domains of the form X.razor2.cloudmark.com, where X is a character of the list (a, b, ..., z). The characters are used in ascending order and this loop stops as soon as the address resolution fails, indicating that no more discovery servers are available. All resolved IP addresses build the set D.

(2) The client connects to a server in D. If the connection fails, a connection to another discovery server in D is tried to be established. If no connection attempt is successful, the system fails.

(3) The client requests a set of *catalogue servers* (C) and *nomination servers* (N). Catalogue servers are used to check if a message is spam; nomination servers are used to report and revoke messages.[7]

(4) D, C, and N are stored on the client and are regularly updated, for instance once a week.

In the first step, several discovery servers are searched for. Once they have been found, several catalogue and nomination servers can be looked up. That is, the Razor network has a replicated structure which makes it difficult to perform a DDOS attack against the servers.

---

[6] The client of the Razor filter is licensed under the open-source Artistics License. The server, however, has not been published.

[7] Note that it is undocumented how and when entries reported to the nomination servers are shared with the catalogue servers.

Filter Engines

The Razor filter contains two different *engines* used to identify similar emails. The *Whiplash* engine is a URL-based filter, which we have already described in Section 3.6.1. The *Ephemeral* engine is a fuzzy hash-based filter that works as follows:

(1) The client connects to one of the catalogue servers in C.

(2) The server sends a seeding number s for the random generator used in the client.[8]

(3) The client picks two disjunct windows of the email body based on the seed s and hashes their characters.

(4) The client checks if this hash value is known to the server it is connected to.

(5) The server's reply contains the classification (spam/ham) and also a confidence value that is derived from the number of reports and revokes and the trust values of their associated users.

A message is considered to be spam if the classification sent from the server is spam and if the confidence value exceeds a threshold value which can be adjusted by the user.

To report a message as spam, the entire spam message is usually sent to a nomination server. For a revoke, only the locally computed hashes are transferred.

Trust Evaluation System

The *Trust Evaluation System* (*TeS*) is the server-based trust system of the Razor network. Only little information is known about TeS since no details have been published—neither about itself nor about the server code. According to [74], the TeS is responsible to determine which hash values that have been reported to the nomination servers are forwarded to the catalogue servers. For this, it is obvious that the TeS has to take those users more serious who have reported or revoked messages in compliance with a majority of other users. Those users who did not assent to the majority in the past will only have little influence on future decisions. However, the exact algorithm of the TeS is kept as a trade secret.

[8] The motivation to receive a seeding number from the server is to allow for changes in the client-side algorithm. However, in the last two years, the seeding number has never changed. We assume that it is very simple for spammers to adapt to a new seed. Therefore, changing it would only be a short-term solution. Another explanation is that once the seed has changed, all known hashes stored on the servers would be rendered useless, as they cannot be matched with the Ephemeral algorithm using the new seed.

Remarks on the Implementation

Re-implementing the Perl source code of Razor was not a trivial task. First, the code contained some mistakes which we, although knowing better, could not fix in the Java code, as our solution would otherwise have become incompatible with the original code. Second, we discovered some flaws in the code which we reported to the Razor community and which have been fixed since then. Particularly, until October 2004 revoked messages have been sent in clear text to the nomination servers. That is, messages that a user considered ham—and therefore of private nature—were sent to the Razor network, where they probably were stored for later usage. See [136] for more information. And third, Razor usually runs on Unix systems and depends on the native `drand48()` method of the GNU C Library, which is not available in Java. Particularly, we had to implement a compatible version of the IEEE 754 floating point standard [118] in Java.

3.7.1 Related Work

The implementors of Razor founded the company Cloudmark, which distributes commercial anti-spam products, for instance the Cloudmark Desktop for client-side users [105]. In addition to the Whiplash and the Ephemeral engines, the Cloudmark products use several other fingerprinting algorithms, which are however not yet supported by the Razor client.

The Pyzor [134] project was initially started in order to provide a Python implementation of the Razor filter. However, since Razor has not published the source code for their servers, the author of Pyzor decided to build a new filter; the Pyzor client and servers are licensed under the GPL and are hosted on SourceForge. Pyzor uses a similar hashing algorithm as implemented in the Ephemeral engine of Razor. Since the fingerprints are not compatible, the Pyzor network is distinct from the Razor network and runs its own servers.

The Distributed Checksum Clearinghouse (DCC) filter [137] incorporates three fingerprinting algorithms. The *body* algorithm just hashes the complete body of an email; only equal messages will therefore match. The other two algorithms, *fuz1* and *fuz2*, calculate different *fuzzy* hashes over parts of the message body. They match even for emails which have slightly been modified but share a common origin. The concrete implementation of the DCC filter has been published as open-source. However, the code has intentionally been written in "ugliest," uncommented C such that spammers cannot easily conceive the details.

NiX Spam [124] is another filter that calculates a fuzzy hash of an email. For this, it first cleans the body of an email by employing deobfuscation techniques and removing HTML tags. The purified text can then be hashed and compared to other fingerprints in the database. The filter is written in Perl and is available as open-source.

Razor, Pyzor, DCC, and NiX Spam can easily be integrated into server-side anti-spam filter systems such as SpamAssassin. We have compared the quality of the former three and present the results in Chapter 5. We have not been able to analyze NiXSpam as the authors provide only a Procmail script but no client application, which we need in our framework to extract the calculated fingerprints.

3.8 Comha

The Collaborative Multi Hash (Comha) filter has been inspired by the work of Zhou et al. [99]. The general idea is to select several text parts of an email as fingerprints. In this scenario, emails are defined to be similar if the number of matching fingerprints exceeds a given threshold. A more detailed description is as follows:

(1) Move a sliding window of width w over the entire text of an email and compute for each position a hash value of the selected text.

(2) Choose the largest[9] n distinct hash values and store them in the fingerprint set F.

(3) Send F to the Comha server.

(4) The server determines all matching fingerprints by querying its report and revoke databases. For each *matching rank* $m = 1..n$, it returns the value of matching emails for both reports ($\#emails_m^{rp}$) and revokes ($\#emails_m^{rv}$).

(5) The classification on the client is determined as one of four cases, tested from (a) to (d):

   (a) **Ham**, if the highest matching rank of reports is below a threshold t.

   (b) **Ham**, if the highest matching rank of revokes is greater than the highest matching rank of reports.

   (c) **Spam**, if there are no revokes in the highest matching rank of reports.

   (d) **Ham or spam**, if the highest matching rank of reports and revokes are equal. The result is calculated as $r = \frac{\#reports}{\#reports + c \cdot \#revokes}$, where c denotes a weighting factor that reflects the importance of revokes compared to reports. If r is greater than or equal to 0.5, the message is considered spam, otherwise it is ham.

[9] We could also choose the smallest values or the values that are nearest to any other number. To increase the robustness of the fingerprints, in the sense of fewer collisions, it is also possible to combine different selections.

From the results presented in Chapter 5 and from empirical analysis, we have derived the values of the parameters in the deployed Comha filter as $w = 20$ and $n = 10$. Furthermore, the parameter t is set to 4 and c to 2 by default; both values can be adjusted by the user. Note that it is not possible to change w, as this would render all calculated fingerprints futile. To support other values of w, it would be necessary to maintain a database for each value of w. Choosing a different value for n would be possible, but we decided to fix its value to reduce the handling complexity.

Moreover, it is possible to specify the minimal number of reported emails that are necessary not to classify a message as ham in step (5). By default, this value is set to 2. This means that at least 2 reports must be sent to the server for the highest matching rank; if there is only one report for the highest matching rank, the next lower matching rank is considered.

The Comha filter also incorporates the Trooth system to ensure that no user can maliciously subvert the filter's integrity. For this, also the trust values of the reporters and revokers for the highest-ranked matches are considered in step (5b) to (5d) of the algorithm. That is, for example for case (5b), only if the sum of all revokers' trust values exceeds a threshold the message is considered ham.

3.8.1 Related Work

As mentioned before, we have taken some ideas from the work of Zhou et al. [99]. They describe a collaborative spam filter which stores the fingerprints in a distributed hash table (DHT) rather than in a central database. Their approach is interesting, as a peer-to-peer system which underlies the DHT is more robust against DDOS attacks than our server-based system. However, managing trust relationships in a peer-to-peer system is more complicated.

Manber describes the arguably first approach to find similar files with this technique in [63]. Schleimer et al. describe their *Winnowing* algorithm in [84], which is also an n-gram-like technique. Muthitacharoen et al. [67] propose an algorithm specifically designed to find similar files in a distributed file systems; this technique has been improved in [31].

Chapter 4

The Trooth Trust System

I NEED TRUTHFUL PERSON IN THIS BUSINESS BECAUSE I DON'T WANT TO MAKE MISTAKE;
I NEED YOUR STRONG ASSURANCE AND TRUST.
(Dr. Rane Jack <dr_ranejack2003@yahoo.co.uk>, 4/4/2006)

In the previous chapter, we described several collaborative spam filters. When a user reports a spam message, *similar* emails are automatically eliminated from other users' inboxes. However, it remains the question why a user should (implicitly) allow other, *unknown* users to remove messages from the own inbox. What if the other user is "a bad guy" or a spammer revoking all the Viagra messages? What if the other user deems all the beloved AOL newsletters to be spam?

In this chapter, we tackle these questions introducing the *Trooth* trust system [5]. In the Trooth system, a user does not blindly trust all other users' decisions. Instead, one learns over time which users are generally assenting with the own opinion about what is spam and what is not. A user considers only those spam reports that were sent by *trusted* users.

In the next sections, we first describe Trooth in a general setting. We examine a *voting scheme* where users want to evaluate arbitrary *items* to be either "good" or "bad." When a user has selected an item, she can either classify it manually or use the Trooth assistance to evaluate it. Assessing it manually means to buy a product, install and experience an application, or read an email in order to check whether it is spam or legitimate. Afterwards, the user casts a vote for the item expressing her opinion to the Trooth system. Alternatively, Trooth can automatically evaluate an item. For this purpose, Trooth analyzes the user's own and other users' *previous votes* in order to *predict* the user's view of the current item. Obviously, the automatic approach should be preferred since it means less effort for the user. But it also involves some risk relying on calculated values instead of the own reason.

After this general introduction, we show in Section 4.5 how Trooth is employed in the Spamato system to secure the collaborative activities of the Earlgrey and Comha filters.

4.1 Related Work

Trust systems are related to the domain of *collaborative filtering* techniques [151]. Generally, a sparse $m \times n$ matrix for m users and n items is considered, in which only a few entries reflect the users' opinions on the items. The goal of a collaborative filtering system is to ease the task of manually choosing a new item by automatically recommending suitable items to users who have not rated them previously. For this purpose, this technique derives future decisions from assenting opinions in the past.

We adopt this notion in that we also assume users to vote for items, *votes* being either "good" or "bad." Additionally, we explicitly introduce *trust values* between users; a higher value denotes more confidence in a user. By this, we implicitly define *special interest groups* which contain users with assenting opinions. In contrast to collaborative filtering, we are not particularly interested in recommendations for arbitrary items—instead, our system computes *evaluations* of specific items, being either "good" or "bad" (or "unknown" if an item cannot be evaluated). Furthermore, we assume a continuous stream of items, which a varying number of users want to have assessed. Thus, the number of users and items is not predefined or bounded to any m and n respectively.

All work on collaborative filtering shares the concept of predicting future user behavior or recommending suitable choices based on historical data. One of the earliest work on collaborative filtering systems is Tapestry [34] which uses a SQL-like language to filter manually annotated (electronic) messages. The focus of current research lies in different areas. Several approaches are described in [54, 42, 9]; a comprehensive research bibliography can be found in [128, 145].

We share some ideas of collaborative filtering but focus on practical aspects. For this purpose, we only provide a heuristic to evaluate an item for a user. We do not provide mathematical analysis as in [50, 9].

In contrast to *offline* algorithms, such as [50, 83, 54], which usually recommend a single item to a user based on a fixed set of votes, we consider a continuous stream of items which a user has to assess. These items can arrive at any time, making predictions time-dependent. Furthermore, users cannot be asked to *train* a server-based system in order to increase the rate of correct predictions; items are always selected client-side. We also store only a minimum of data on the server and evaluate items on clients. We differ from other *online* approaches, such as [27, 10], in that we do not consider a round-based synchronous model. Again, in Trooth users can vote for items at any time.

Systems recommending articles to clients, such as employed by Amazon [59], differ from ours in two aspects. First, Amazon chooses a fast, server-based approach while we explicitly integrate clients in the evaluation process of items. And second, we do not recommend items to users but evaluate

them when a user is confronted with them, for instance when a new email arrives.

Other collaborative spam filter system also rely on trust systems to distinguish between trustworthy and malicious users. The Razor [135] filter and the Cloudmark products [105] take a strict server-based approach [74]. In the SpamGuru [88] system, a server-based, supervised clustering of items or rather users might be employed.

4.2 Preliminaries

In this section, we describe some basic aspects which we use and refine in the next sections. The general aim of Trooth is to help rate items, such as products, people, or emails, to be either "good," "bad," or "unknown" by allowing users to classify the item as "good" or "bad."[1] Given an item, a user first tries to revert to a calculated recommendation. If the evaluation is "unknown," the user has to manually assess the item. Afterwards, she casts a vote, sending her manual assessment to the Trooth system.

In this section, we assume that a pre-defined, globally valid evaluation for an item exists, which has to be exposed for each item. In Section 4.3, we revise this assumption and instead calculate *individual, user-specific opinions* about each item.

4.2.1 Evaluation Functions

A global evaluation of an item can be derived from all *user votes* in various ways. For instance, using a simple *majority* evaluation, the overall categorization of an item is "good" if a majority of all users ($> 50\%$) votes in favor of the item, "bad" if a majority votes against the item, and "unknown" if the number of "good" and "bad" votes are equal. For the more general *threshold* evaluation function, we classify an item to be "good" ("bad") if the ratio between "good" ("bad") votes and all votes is greater than a threshold value h_g (h_b) and "unknown" otherwise.

It is easy to extend this *simple* scheme from the set {good, bad, unknown} to a more general range where votes and evaluations can be in the interval $[0, 1]$. In this case, we can define the result of an evaluation to be the average of all vote values. This is however not necessary in the Spamato system where a user always votes for spam (bad) or ham (good).

4.2.2 Weighting Votes With Trust Values

So far, votes or rather users have been considered to be equally important. In real life, however, it can be beneficial to apply different weights to votes

[1] We assume that a user who does not know how to classify an item does not vote at all rather than voting for "unknown."

or, in other words, to consider some users to be “more equal than others.” Since we assume the existence of a single, pre-defined evaluation for each item, users who often agree with the majority of users should be trusted more than those who regularly dissent.

Ideally, this means trying to separate users into two groups: One group contains those users who are *trustworthy* and the other group those who are *malicious*. Practically, it is possible to approximate these groups by introducing *trust values* for each user that are adjusted whenever new information is available. Then, instead of simply summing up equal “good” (and “bad”) values as before, each vote is previously weighted with the trust value of the associated user. For this approach to make sense, we generally assume that the group of trustworthy users are a majority, or more precisely: that those users who agree with the majority are trustworthy.

The *Additive Increase, Multiple Decrease* (AIMD) approach takes user specific and automatically adjusted trust values into account. When all users have cast their votes, the trust values are modified. Using AIMD, users who voted *correctly*, that means in accordance with the majority, are awarded by slightly increasing their trust values. On the other hand, users whose votes do not comply with the majority are punished by harshly decreasing their trust values.

Note that we are rating in two different domains: On the one hand, we want to evaluate items by having unknown users vote “good” or “bad” for it. On the other hand, we want to calculate trust values for unknown users to make their votes more reliable. The voting (and thus also the evaluation) is actively performed; trust values are implicitly generated. While in principle it is possible to let users choose whom they want to trust, in reality this is considered too involved.

4.2.3 Implementation Issues

For applications like Spamato or rather its collaborative spam filters, the voting for an item (in this case, reporting or revoking a misclassified email) and its categorization (spam/ham) will not take place at a single point in time. Instead, users can always vote for an arbitrary message, and Spamato classifies a message whenever it arrives in an inbox. Additionally, not all users vote for all items, since not all users receive the same messages. Therefore, user votes have to be stored for later usage and the evaluation of an item has to be recalculated whenever a new vote has been cast. In other words, evaluations are time-dependent. Furthermore, users must not be able to vote more than once for the same item or multiple votes have to be handled in a reasonable way. Finally, users have to be authenticated in order to prevent manipulations.

Implementing the AIMD approach or weighting algorithms in general entails some difficulties since users can vote at any time. The question is:

"When should trust values be updated?" On the one hand, new trust values can be calculated at a single point in time, for instance after a specific number of votes were received. For this approach, only few data sets have to be stored on the server. It also saves server resources as trust values are updated only rarely. However, this approach obviously ignores later votes and thus important information to provide complete and fair trust values. On the other hand, trust values can be calculated whenever a new vote is received. As it is possible that new votes for an item change its evaluation, the trust values of all involved users have to be updated. For this, extensive historic information about the whole voting process has to be managed on the server. In both cases, the server has to store the trust values for each user and the overall evaluation of each item—to avoid time- and resource-consuming calculations whenever these values are required. In summary, the second approach demands for significantly more server resources than the first solution.

4.3 The Trooth System

In this section, we introduce Trooth as a robust, partially decentralized, collaborative, and personalized voting and trust system. It is robust as it withstands malicious users who are trying to cheat the system, partially decentralized as clients are explicitly involved in the voting and evaluation processes, and collaborative and personalized as users interact with each other for collective benefits.

In the previous section, we have assumed that it is possible to globally evaluate an item—that an overall evaluation exists which coincides with the votes of all trustworthy users. But the separation of users into groups of trustworthy and malicious users often is too harsh. In fact, the assumption that an objective *overall* categorization can always be calculated is arguably wrong.

We believe it is more reasonable to individually evaluate an item for each user separately. A user does not distinguish between trustworthy and malicious users anymore, but between users who generally vote in accordance and those who do not. Thus globally seen, users are implicitly separated into several *special interest groups* who share a *similar opinion* rather than to discriminate them with the "black & white" scheme described before.

Note that we still believe that trustworthy and malicious users exist. While the former describe users who really try to express their opinion, the latter usually vote against the common sense and try to deceive the system, probably for personal benefits. We also consider users who make mistakes and others who just do not understand how to operate a voting system. But generally, from a user's point of view, all these categories can be reduced to assenting and dissenting users only. For the sake of simplicity, in this section we use the term *malicious* also for incautious and unaware users.

Depending on the voting domain, the number of groups can vary significantly. Although we expect groups to be rather large and overlapping, in the extreme, each user might trust only herself so that the number of groups equals the number of users. But this especially expresses the strength of the system: Even if all except one user are malicious, this one will (eventually) figure out not to trust anybody except herself. Thus, the system can even serve different minorities with satisfying results while approaches that assume the existence of an objective evaluation cannot.

Since Trooth does not compute a global evaluation of an item for all users, it is possible to reduce the consumption of server-side resources to a minimum. Therefore, in Trooth, we store only (item,user,vote)-tuples server-side and calculate user specific trust values client-side.

4.3.1 Managing Votes and Trust

As in structured peer-to-peer systems, we assign each item and each user a unique identifier from an interval $[0, ..., N]$, organized as a "ring." Thus, we can use the notions of *clockwise* and *anti-clockwise* to denote neighbors on the ring. In Section 4.4, we show how user IDs (and signatures to authenticate users) are generated in the Spamato system.

The Voting Process

When a user votes for an item, she sends her opinion ("good" or "bad") to the Trooth server and locally adapts the trust values for other users who voted for the same item. In more detail, the voting process takes the following steps:

- User u_0 sends a vote v_0^i for an item i to the server where the (i, u_0^i, v_0)-tuple is stored.

- The server assembles two lists which are populated with the identifiers of other users who previously voted "good" (list G) and "bad" (list B) for item i. Each list contains a maximum of k user IDs that are numerically nearest to u_0 in respect to the ring formation. The lists G and B are sent to the client.

- User u_0 locally adapts the trust values of the users sent to her by increasing the trust values of those users who agreed with her own vote v_0 and decreasing the trust values of those users who voted against it (using the AIMD approach mentioned before or any other weighting scheme).

The Evaluation Process

To classify an item, "good" and "bad" votes from the server are weighted with the client-side stored trust values. In detail, the following steps are performed for the evaluation process:

- User u_0 sends a query for item i to the server.
- The server returns two lists containing identifiers of users who voted "good" (list G) and "bad" (list B) as in the second step of the voting process described above.
- User u_0 extracts the $l \leq k$ most trustworthy users of each list, resulting in the lists $G' \subseteq G$ and $B' \subseteq B$.
- Finally, the classification can be calculated using the threshold evaluation function and the votes weighted with the trust values of the users in G' and B' as parameter.

User Specific Parameters

In the Spamato system, the size k of the "good" and "bad" lists returned by the server, l that denotes the number of the most trusted users to select from the lists, *inc* and *dec* as parameters of AIMD, and the values of h_g and h_b for the threshold evaluation function are user specific values. Thus, the user is able to further configure the processes to some extent on the client-side.

Discussion

In the second step of both algorithms, the server returns about $k/2$ clockwise and anti-clockwise neighbors of user u_0 for each list. If there are less then k users who voted "good" or "bad" for item i, we return only that many without any loss in quality. Assuming that users usually vote for the same *type* of items, it is reasonable to believe that the total number of trust values that have to be handled client-side is bounded.[2] This means that each user generally stores only a small subset of all users who share the same opinion. It is also an advantage that a user's vote can only affect those users in the implicit neighborhood. Thus, the impact of a possibly malicious user trying to cheat the system is limited. On the other hand, though, the rather high consumption of bandwidth for each voting and evaluation operation can be regarded as a drawback.

By choosing only the most trustworthy users in the third step of the evaluation process, we decrease the influence of unwanted users to a minimum.

[2]Regarding emails, users who are contained on the same email address list (compiled by spammers) often get the same spam messages.

As an optional step, trust values could be adjusted after the evaluation process similar to the last step in the voting process. Doing so would amplify the influence of trust values even more. Additionally, the calculated evaluations could automatically be sent as a personal vote to the server. If the user does not agree with the evaluation, she would (immediately or after some time) send her correct opinion rejecting the old one.

Note that a malicious user who tries to gain a high trust value in order to manipulate the evaluation process, previously has to "play by the rules" for a long time, thereby helping other users more than harming the system. Moreover, to manipulate a particular user (or a group of users with assenting opinions), it is necessary to, first, get an ID that is near that of the user, and second, to know which items the user is "interested" in. While attacking one particular user is hard, it is a futile attempt to oppose against many groups or even all users at once. Thus, Trooth significantly reduces the impact of malicious users in the evaluation process.

4.3.2 The Majority Heuristic

We introduce the *majority heuristic* that can be applied as a special case when *many users* almost *unanimously decide* about an item. That is, in contrast to what we have described so far, the evaluation of an item is determined by the majority of users without regarding any trust values. To use it safely, one has to rule out the chance of malicious users being a majority.

In the *voting process*, the server still stores the vote of a user for an item. But it does not return any data and the client, therefore, cannot adjust any trust values. In the *evaluation process*, the server sends only the number of "good" and "bad" votes to the client. Therefore, the client is not able to select her most trusted users anymore; all votes count the same. The evaluation for the item is calculated using the majority or threshold evaluation function.

The majority heuristic clearly simplifies the voting and evaluation processes by transferring less information between client and server, and thus also saving bandwidth. However, this approach slightly reduces the reliability of the classification since trust values are not considered anymore. But this can be neglected because there are almost no dissenting votes, and malicious users cannot be a majority as defined above.

4.3.3 Extending Trooth

In this section, we provide two extensions for the Trooth system which implement orthogonal ideas. While the first one describes a system which centers all activities on one server, the second sketches the idea of a completely decentralized approach.

Server-Side Trust Values

In the Trooth system described so far, each user stores a list of trust values on the client. As we previously explained, although it is possible that this list contains a trust value for all other users, it is more likely to hold only a small subset of neighbors. In contrast to our earlier motivation, we could store trust values and determine the classification of an item solely on the server. By sending the parameters of the voting and evaluation processes from the client to the server, the user would still be able to adjust the outcome as before. Therefore, running Trooth server-side implies no restrictions for the user.

Having trust values stored on a single server also allows to *globally* analyze this data. Consolidating the trust values of each user could disclose a variety of interesting information—for instance, how groups of assenting users are organized or whether malicious users or rather users whom nobody trusts exist. Another advantage is that users can now share their trust values between different machines or accounts. Furthermore, aggregated trust values could be used to provide a new user with some initial data. On the other hand, processing Trooth on a server drastically increases the resource demand for CPU and storage (while the bandwidth consumption is lowered).

We want to emphasize that a server-side Trooth system is not similar to approaches summarized in Section 4.2. The main difference is that we manage trust values for each user separately, still assuming that it is more promising to rely on several groups of assenting users than on trustworthy and malicious (in its original sense) ones.

Distributed Trooth

Trooth shifts most of the work to the client, keeping only the storage of (item, user, vote)-tuples on the server. This is a good foundation to completely decentralize the Trooth system.

We propose the usage of a distributed hash table (DHT) such as Chord [90] or Kademlia [64] to obtain a server-free Trooth system. In such systems, the "lookup" operation is the most important command which maps a *key* to the peer being responsible for it. Besides this, the "store" operation stores the *value* associated with a key at the managing peer.

Regarding Trooth, the information about user votes have to be managed in the DHT. In the voting process, (item,user,vote)-tuples are the values to be stored at the peer responsible for the item. For this, the mentioned DHTs have to be adapted only slightly to support the storage of multiple values for one key. Similarly, in the evaluation process a user performs a lookup for an item and the responsible peer has to return a subset of all votes that have previously been cast for it. Thus, a fully decentralized Trooth system can generally be realized.

However, there are some difficulties. The voting and evaluation processes will take more time due to the nature of a DHT, where the responsible peer has to be looked up by routing through several intermediate stations. Furthermore, a DHT has to manage other issues such as the handling of joining and leaving peers, counter measurements for hot spots, and caching and replication mechanisms. Although ensuring trust among peers can also be regarded as a key task of DHTs, we want to describe two aspects of this problem in relation to Trooth: *authenticated* and *complete* votes.

In Trooth, we assume the existence of unique user identifiers as well as the possibility to verify which user has cast which vote. In Section 4.4, we describe how we guarantee these assumptions in the Spamato system by generating a public/private-key pair for each user, which is used to sign votes. Although this approach is server-based, we could still employ it for generation and verification of keys only. For a pure distributed approach, though, it is necessary to abandon this server, too.

Another drawback is that one cannot trust peers. A peer is able to alter information in any way before sending it to a user. Thus, votes can be modified, coined, or deleted at will. While the signing of votes will help to detect modified or coined votes, peers cannot be kept from concealing them. One solution to this problem is to store votes not only at one peer but at many. Similarly, a lookup would have to return votes from several different peers. Although this increases the amount of data in the system and the effort to store and query for it by the replication factor, the reliability of the result will be increased accordingly.

4.4 The Spamato Authentication and Authorization System

The *Spamato Authentication and Authorization System* (SAAS) is used to create unique identifiers for users who want to interact with the Trooth system. Additionally, SAAS generates a public/private key pair with which users (automatically) sign their votes to prevent any cheating and manipulation attempts.

Since Spamato (and therefore SAAS) is embedded into an email client, SAAS can make use of the authentication process between the email client and the server. In other words, if a user is able to receive an email that has been sent to her via SAAS, the user is also sufficiently authenticated for Trooth.

In more detail, the first time Spamato is started, the SAAS client locally generates a public/private key pair. The public part of this key pair and the user's email address are sent to an SAAS server (using a TCP connection) which in turn sends a random *challenge email* to the stated address. On receiving this challenge email, the client signs the message with its private key

and sends it back to the server (again using a TCP connection). Thereafter, the user is fully registered with the SAAS server which stores the user's public key and the (hashed) email address as an identifier for the user.

The actual implementation is slightly more sophisticated to allow for a reregistration of users who want to use the same SAAS account. Additionally, the Trooth and SAAS servers need to exchange data so that the Trooth server can validate a user's signature. In the Spamato system, the user information is stored by SAAS in a database that can also be accessed by Trooth.

4.5 Applying Trooth to Spamato

The Earlgrey filter is a collaborative URL filter (see Section 3.6). Upon receiving a new message, it collects all URLs in the message, extracts the domains, and calculates a hash value of them. This hash value is sent to the Earlgrey server which queries a database to find out whether the message is spam or legitimate. Entries in the database are collaboratively inserted by users who report "spam" or revoke "legitimate" emails (or rather the calculated hash values). Thus, users help each other to filter spam messages. A similar approach is taken by the Comha filter as described in Section 3.8.

Since not all users define the term "spam" equally—some also declare unwanted newsletters to be spam while others like to read about online drug stores—clearly, a system like Trooth is necessary to handle these different opinions. In the context of Trooth, the hash values calculated by the collaborative spam filters are the identifiers for items, users are identified with their email addresses (or their SAAS public keys), and reports and revokes correspond to "bad" and "good" votes, respectively. As said before, to prevent malicious users from harming the system, votes are signed with a user specific private key. Additionally, the Earlgrey and Comha servers ignore multiple reports/revokes and removes contrary votes for the same message and user.

Although Trooth performs well securing the collaborative activities of the Earlgrey and Comha filters, it has one problem which cannot completely be solved. Trooth assumes unique item identifiers. However, as fingerprinting algorithms are not perfect, undesired collisions of spam and ham hashes cannot be eliminated. That is, a user reporting a spam message identified by the fingerprint "X" might not be trusted by a user who receives ham messages which are also mapped on the fingerprint "X". For instance, a legitimate newsletter containing the URL `http://www.amazon.com` would conflict with a spam message that (maliciously) downloads images from the same domain. A possible solution is to use whitelists to reduce such collisions; but this solution can only be applied if one is aware of the problematic fingerprints. In real life, fortunately, this weakness seems to be negligible as the Earlgrey and Comha filters have not exhibited any detriments so far.

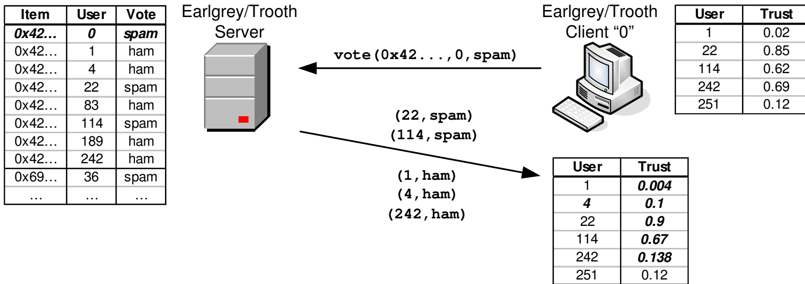

Figure 4.1: This figure exemplifies the voting process. User "0" classifies the message "0x42..." as spam and adjusts her trust values.

4.5.1 Example

Figure 4.1 exemplifies the voting process. For this example, we set $k = 3$, $inc = 0.05$ and $dec = 0.2$ (AIMD parameters), and the user identifier space is in the range of 0 to 255. First, client "0" sends a spam report for the email identified by the hash "0x42..." to the Earlgrey/Trooth server. The server inserts the vote into the voting table and populates two lists for spam and ham votes of users whose identifiers are numerically nearest to 0. Since only two users have classified the message as spam, the spam list is returned with these two entries only (22 and 114), while the list of ham votes contains three entries (1, 4, and 242). Next, user 0 adjusts the trust values using the AIMD approach. Since she voted for spam, users 22 and 114 are awarded, and users 1, 4, and 242 are punished by increasing or decreasing their trust values, respectively. For instance, user 22 has an old trust value of 0.85 which results in a value of 0.9 after increasing it by $inc = 0.05$. Similarly, user 242 has an old trust value of 0.69 which is decreased to 0.138 after multiplying it with $dec = 0.2$. User 4 has not been in the trust table before. Therefore, she is rated with a default value of 0.5 before being punished.

In Figure 4.2, an example of the evaluation process is depicted ($k = 3$, $l = 2$, $h_g = \frac{2}{3}$, $h_b = \frac{1}{3}$). After sending an evaluation request for the message identified by the hash "0x31..." to the server, user "0" receives two lists as described before. The client extracts $l = 2$ votes of each list which have been cast by users she trusts most (189 and 242 for spam, and 22 and 114 for ham). Again, one user (189) has been unknown and was therefore rated with the default value 0.5, which was chosen since this value is higher than the third spam choice (user 1 with a trust value of only 0.004). The trust values are accumulated and the evaluation is performed using the threshold function. Since the ham votes are more trusted (1.57 to 0.638), the overall classification for the email is "ham."

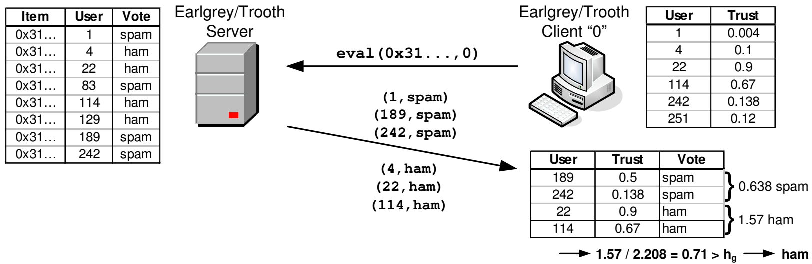

Figure 4.2: This figure exemplifies the evaluation process. User “0” requests the votes for the message identified by “0x31...” and evaluates it as “ham.”

4.6 Concluding Remarks

We have been using Trooth in combination with the Earlgrey filter for about two years without significant problems (Comha was added later). We have to store only little information on the server. And as the evaluation is completely executed on the clients, the load on the server is reduced to a minimum.

Currently, the biggest disadvantage we see is that we do not have any usage statistics about the Trooth system. We do not exactly know which users trust or distrust each other. This information would be interesting to analyze on a large scale, as it might indicate whether our assumption that user groups of similar opinions really exist. On the other hand, this data might probably be too insightful, as it reveals personal preferences, which users might not be willing to share. Nevertheless, gaining a better understanding of the collaboration among users is a fascinating domain to study and can help improve trust systems further.

Chapter 5

Analysis of Collaborative Spam Filters

> `It is an honor to be working in collaboration with FGS`
> `because they are a group of people who really want to make a difference`
> `in the fight against crime.`
> `(Inside Penny Stocks <Trudy@comcast.net>, 2/26/2006)`

In this chapter, we evaluate collaborative spam filters on a set of collected emails in an offline experiment. This "clinical" study makes it possible to analytically compare different fingerprinting algorithms, as they are tested on the same email corpus.

Several other studies have been conducted in literature, mainly analyzing statistical spam filters [7, 65, 22, 21]. When checking an email, statistical filters usually provide a spam probability reflecting the confidence in their classification. The spam probability can be used to generate ROC graphs [29] that visualize the trade-off between spam and ham misclassifications. In contrast, we evaluate fingerprinting algorithms, which produce a binary classification (*match* or *no match*) rather than a confidence value. Therefore, a ROC analysis cannot be applied to our study.

Instead, we define a set of performance criteria in Section 5.2, which we use to evaluate the fingerprinting algorithms described in the next section. In Section 5.3, we present the results for a manually categorized corpus; Section 5.4 contains the results for the TREC corpus. Finally, we discuss our study in Section 5.5.

5.1 Fingerprinting Algorithms

We analyzed the following collaborative spam filters previously described in Chapter 3: Comha[1], Earlgrey, DCC, Pyzor, and Razor. The Razor filter

[1] The Comha filter was used with a window size of 20, a selection of the 10 largest hash values, and a minimal number of 4 matching hashes. We experimented with other

contains the hash-based Ephemeral and the URL-based Whiplash engines; we assessed both of them separately and also a combined version (Razor.multi) for which only one of them had to match (OR-combination). DCC has three algorithms: Body, Fuzzy1, and Fuzzy2; again, we used them separately as well as in an OR-combination (DCC.multi).

Additionally, we experimented with the following fingerprinting algorithms. If not otherwise explained, a match for a hash-based technique is given if two fingerprints (for the same algorithm) are identical—with the exception that an "unknown" fingerprint never matches any other fingerprint. Note that comparing fingerprints that are not hash-based can be very expensive in terms of the number of necessary comparisons. That is, implementing for instance the *Alpha* algorithm on a server is from a practical perspective rather futile. Nevertheless, we also present the results for such algorithms below.

- **Hash.start**: If an email contains more than 100 characters, this algorithm hashes the window starting at index 10 and ending at index 59 (denoted as position [10..60]). If an email contains 100 or fewer characters, the fingerprint is "unknown."

- **Hash.end**: If an email contains more than 100 characters, this algorithm hashes the window for the position [text.length-60..text.length-10]. The fingerprint is "unknown" if 100 or fewer characters are contained in an email.

- **Multihash**: If an email contains more than 1000 characters, this algorithms combines three hash values; otherwise, the fingerprint is "unknown." The first hash value is calculated for the window at position [100..200], the second hash value is calculated for the position [450..550], and the third hash value is calculated for [text.length-300..text.length-200]. The AND-combination of the three hashes is referred to as *Multihash.and* and the OR-combination as *Multihash.or*.

- **Alpha**: The fingerprint of this algorithm is a hash table that contains letters as keys and their frequencies as values; the idea has been taken from [66]. The fingerprint is "unknown" if 10 or fewer characters are contained in an email.

  When comparing two fingerprints (F_1 and F_2) with each other, we calculate the difference of their frequency tables. We also determine two threshold values T_1 and T_2 that depend on the length of the emails for which F_1 and F_2 are calculated. Particularly, we have chosen the threshold values to be $T_{\{1,2\}} = 0.1\cdot \text{text}_{\{1,2\}}.\text{length}$.[2] If the difference

values and found these to perform best in our scenario.

[2] We also experimented with different fractions but have chosen 0.1, as the results were best with this value.

is less than T_1 or T_2, the fingerprints are defined to be similar. Note that the comparison function is symmetric but not transitive.

For example, if F_1 contains 100 'A' and 200 'B' and F_2 contains 95 'A', 206 'B', and 20 'C', the difference is $5 + 6 + 20 = 31$. F_1 contains 300 characters, therefore, T_1 is 30; similarly, T_2 is 32.1. Since $31 < 32.1$, F_1 and F_2 match.

As a generalization of this scheme, we tried an n-gram-based approach for which we did not count characters but hashes over small windows of size 2 to 5. We also mapped the calculated hash values to a fixed number of buckets (8 to 4096) in order to reduce the cost for comparisons. However, the results were worse than for the single character-based Alpha algorithm so we decided to skip these results in the following sections.

- **Alpha.multi**: This algorithm AND-combines three values: the size of an email, the average of the letter frequencies (*avg* criterion), and the standard deviation of the letter frequencies (*std* criterion). Particularly, the lengths of two emails are allowed to differ by at most 50 characters, the avg criterion may differ by 0.02, and the std value by 0.1 percent. The fingerprint is "unknown" if the message size is 10 or less. As in the previous Alpha algorithm, comparing fingerprints regards symmetry but not transitivity.

- **Nilsimsa**: We use the Nilsimsa algorithm as described in Section 3.5.1. However, we have experimentally derived a value of 230 matching bits for good results rather than 152 as proposed by the author. The "improvements" mentioned in [24] could not be confirmed by our experiments. Therefore, we use the original Nilsimsa algorithm. Similar to the Alpha algorithms, the comparison function is symmetric but not transitive.

5.2 Performance Criteria

We assume that a categorized corpus exists in which similar emails—that is, emails that share a common origin but might have been modified by spammers—have been sorted into the same category. A good fingerprinting algorithm should classify emails in the same category to be similar. The more matching fingerprints the algorithm calculates for the emails in each category, the better it is. In order to compare fingerprinting algorithms objectively, we define the following performance criteria that reflect the capability of an algorithm to map similar emails on similar fingerprints.

The *Total False Negative Rate* is calculated as the ratio between the sum of emails not matched and the sum of emails that should be matched; the

set C contains all email categories and each $c \in C$ contains all emails in that category:

$$\text{FN}_{\text{total}} = \frac{\sum_{c \in C} \sum_{e \in c} \text{\#emails not matched by e in c}}{\sum_{c \in C} (\text{\#emails in c}) \cdot (\text{\#emails in c - 1})}$$

The *Average False Negative Rate* is calculated as the number of emails that did not match all other emails in the same category ($\text{\#emails}_{\text{incomplete}}$) divided by the number of all spam emails (#spam):

$$\text{FN}_{\text{average}} = \frac{\text{\#emails}_{\text{incomplete}}}{\text{\#spam}}$$

The *Matched-X False Negative Rate* is calculated as 1 minus the number of emails that matched at least X other emails in the same category ($\text{\#emails}_{\text{matched-x}}$) divided by the total number of spam messages (#spam):

$$\text{FN}_{\text{matched-x}} = 1 - \frac{\text{\#emails}_{\text{matched-x}}}{\text{\#spam}}$$

The $\text{FN}_{\text{matched-x}}$ is of particular interest, as in a collaborative spam filter network a match is usually not announced unless several reports for the same fingerprint have been registered.

So far, we have described performance metrics to evaluate fingerprinting algorithms applied to spam emails of the same kind. Even more important is the capability not to erroneously match ham messages with spam messages. Therefore, we will also measure the *False Positive Rate* that we have defined as the fraction of ham emails whose fingerprints collide with at least one spam email ($\text{\#ham}_{\text{matched}}$) divided by the number of all ham emails (#ham):

$$\text{FP} = \frac{\text{\#ham}_{\text{matched}}}{\text{\#ham}}$$

We also show the *Spam Unknown Rate* that is defined as the number of spam emails for which no fingerprint could be calculated ($\text{\#spam}_{\text{unknown}}$) divided by the number of all spam emails (#spam); the *Ham Unknown Rate* is defined accordingly.

$$\text{Spam}_{\text{unknown}} = \frac{\text{\#spam}_{\text{unknown}}}{\text{\#spam}}$$

We have evaluated other metrics as well. However, as their values are related to each other, we have chosen the subset described above and use it for the evaluation presented in the next sections.

5.3 Results for a Categorized Corpus

We manually compiled a categorized spam corpus to evaluate the fingerprinting algorithms. The corpus consisted of 5005 spam emails, which we had sorted into 213 categories; each category contained at least 5 similar emails. Furthermore, we used a private ham corpus with 5419 ham messages. Note that we did not expected the algorithms to reach optimal "0.0" values. Although we revised the corpus thoroughly, it could still contain a few misclassified messages. Nevertheless, smaller values definitely indicate better algorithms and the performance criteria can thus be used to compare algorithms with each other. The results are depicted in Figure 5.1, on which we comment below.

Original Corpus

The false positive rate was for all filters very low—with two exceptions: For the Earlgrey filter, it was about 11.5% and for the Hash.start algorithm about 6.5%. As we see later, the Hash.start algorithm was very sensitive to HTML mails, as also ham emails often exhibited a standard HTML header. The poor result for the Earlgrey filter was astonishing since the Whiplash engine of the Razor filter should perform similarly. After verifying the results, it was obvious that the Whiplash algorithm did not correctly detect all domains—neither using the original Perl code nor with our Java implementation.[3] We want to emphasize that such high numbers of false positives are usually unacceptable. However, most collaborative spam filters use a whitelist to eliminate obvious misclassifications. In our case, removing only the domain "`ethz.ch`" decreased the false positive rate by about 7%.

The $\mathrm{FN}_{\mathrm{average}}$ rate was generally very high, as only few messages matched all other fingerprints in the same category. The $\mathrm{FN}_{\mathrm{total}}$ was a better criteria since it also considered the number of partially matched emails.

The Comha filter clearly outperformed all other algorithms. Although its false positive rate (1.3%) was higher than for Pyzor and Razor (both almost zero), all other values were significantly better. Interestingly, also the Alpha and Nilsimsa algorithms performed very well—however, finding a match for these algorithms was more expensive, as a fingerprint had to be compared to all other fingerprints (rather than looking it up in a hash table).

[3] We used the original code for our evaluation to circumvent any problems with our own implementation. As we do not use the *url* plug-in but completely re-implemented the original code, it was no surprise that the Java code also contained the now detected domain detection bug.

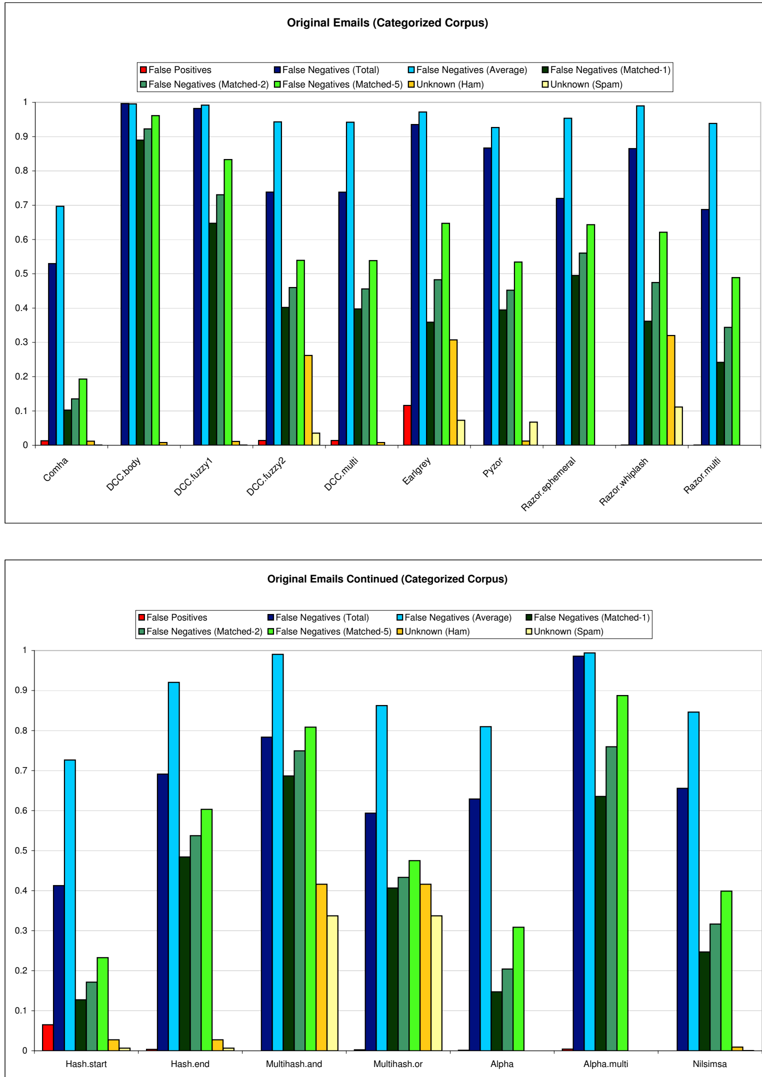

Figure 5.1: Results for the original categorized email corpus.

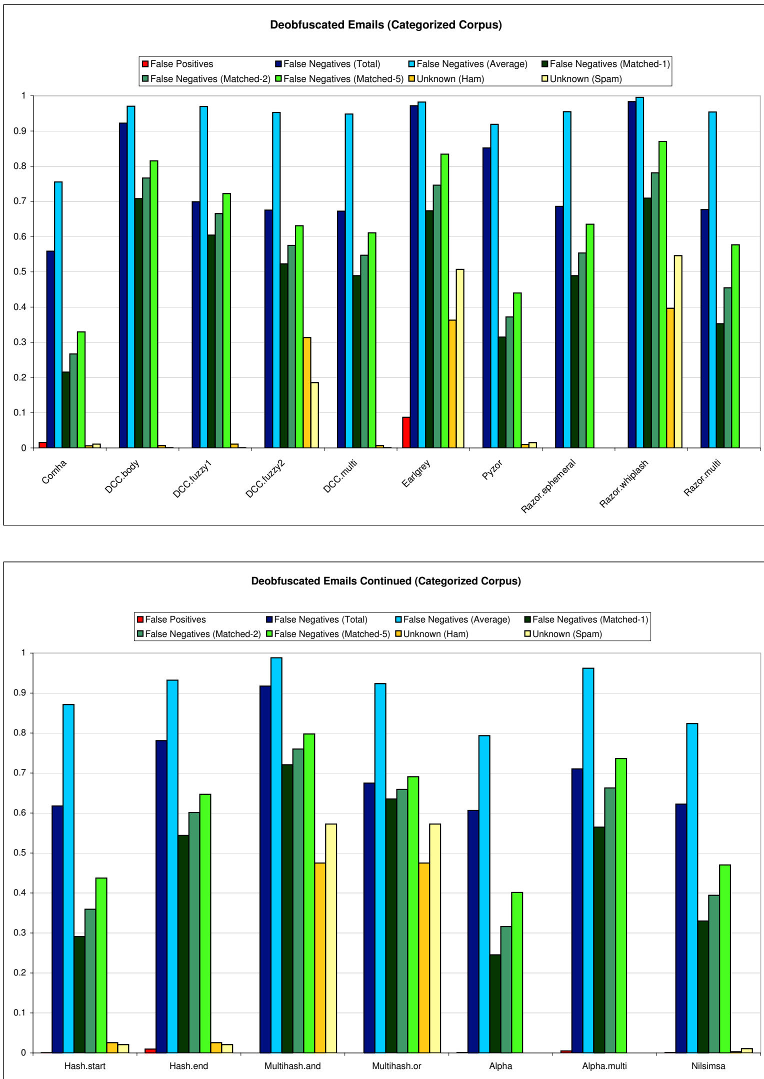

Figure 5.2: Results for the deobfuscated categorized email corpus.

Deobfuscated Corpus

One of our first concerns was that simple filters, such as the Alpha or the Hash.start filter, were good HTML detectors rather than spam filters. Therefore, we used our *deobfuscator* plug-in (see Section 2.4.6) to eliminate some of the worst spamming techniques, and primarily to remove HTML code from the email body. For the *deobfuscated corpus*, we deleted all emails that, after removing the HTML code, had only an empty body. This left 4788 spam and 5329 ham messages for which the results are shown in Figure 5.2.

As can be seen, most filters performed worse in the deobfuscated scenario. Only the Pyzor filter had a significant benefit; also the values for the DCC.body and the Alpha.multi filter improved slightly. We think that most of the other algorithms matched fewer emails because the deobfuscated email bodies contained less information. However, this has to be studied further.

5.4 Results for the TREC Corpus

We chose the TREC 2005 Public Corpus [22] to verify our results on a different email corpus. It is based on the Enron corpus [51][111] and contains about 92000 emails in an almost original, unmodified form. Other corpora, such as the Ling Spam corpus [7] and the PU corpora [8][133], could not be used since they contain only pre-processed emails, lacking header or body information.

After removing emails larger than 100 kB and emails that could not correctly be parsed by our corpus builder tool, 12606 ham and 41960 spam messages were left for our evaluation. Since we did not categorize the spam corpus, the only meaningful experiment we conducted was to measure the false positive rate. Note that we did not deobfuscate the TREC corpus, as the differences on the categorized corpus seemed to be too small to justify the additional effort.

The false positive rates measured on the TREC corpus are illustrated in Figure 5.3. For better comparison, we also show the false positive rates of the original, categorized corpus. Note that the y-axis is only from 0 to 0.12.

As can be seen, the Comha and the Earlgrey filter performed slightly better than before; the Hash.start filter even showed an improvement of more than 4%. The Whiplash engine and the Hash.end algorithm were significantly worse compared to the categorized corpus. Also the Multihash.or, Alpha, Alpha.multi, and Nilsimsa algorithms performed worse on the TREC corpus. Note that the DCC.body, DCC.fuzzy1, Pyzor, and Multihash.and algorithms produced no false positives on both corpora.

It is hard to explain the differences between the results. Analyzing further corpora could help understand which types of legitimate emails are often matched with spam emails.

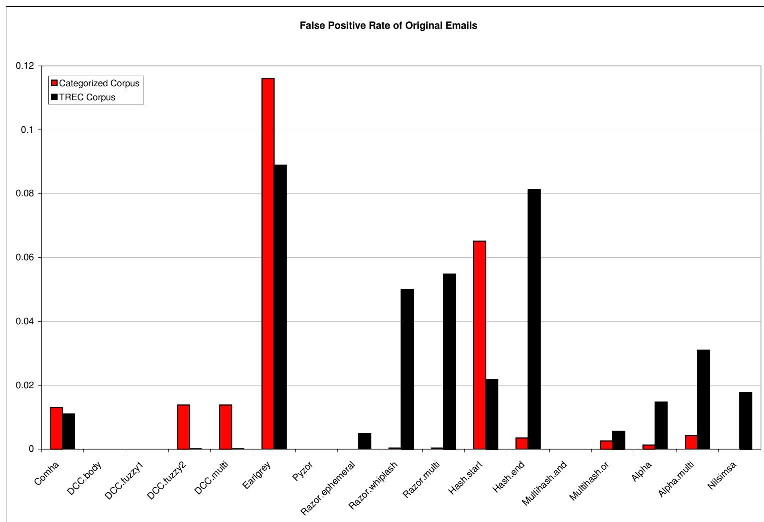

Figure 5.3: False positive rates for the categorized and TREC corpora.

5.5 Concluding Remarks

In this chapter, we analyzed some widely deployed collaborative spam filters and some experimental fingerprinting algorithms. We introduced several performance criteria that we used to evaluate the capability of the algorithms to match similar emails in our categorized email corpus. The results indicated that our Comha filter performed best, but also some of the experimental candidates did quite well. Overall, none of the filter results were particularly poor. However, some of the filters caused a high number of false positives. As mentioned before, we believe that this problem can be alleviated by introducing server-side or client-side whitelists as they are employed by many collaborative spam filters.

Our spam corpus has manually been compiled; it contains about 5000 messages sorted into 213 categories. It would be interesting to verify our study on a larger categorized corpus. However, building such a corpus is very time consuming and error-prone. To get a larger corpus, one could try to apply typical techniques used by spammers to create "similar" emails manually; this has been suggested and employed for example in [99]. To generate a large amount of real spam emails, it would be best to use a standard spamming tool as it might be available on the Internet.

In our experiments, we did not measure the effect of combining filters. For instance, it would be interesting to see how the Earlgrey filter and the Whiplash engine performed in concert, or whether Razor, Pyzor, and DCC could be improved when they were combined with the simple Hash.start or Hash.end algorithm. The *stacking* of classifiers has been studied for instance in [82, 88, 65, 21]; the results show that the combination of different (statistical) filters increases the filtering accuracy compared to the application of individual filters. We see the evaluation of combined fingerprinting algorithms as an interesting future research direction.

Chapter 6

The Plug-In Framework

abidjanstudiousbicycleaustinsienadeliberatefromgerhardperseusschoon=
ermoencalanimadversionlilianabsolvebikeinnovatelagosplanoconcavejacmchroni=
cPLUGINvolute
(Jarvis Carrier <16r3em@front.ru>, 5/6/2006)

With increasing complexity, large software projects tend to get unwieldy, unmanageable, or even out of control. Researchers and developers have tried to counter this crisis with several approaches. Today, component and service orientation, plug-in architectures, and aspect oriented programming are "en vogue." The software development process evolves to a software management process, where more and more *building blocks* are developed separately and connected afterwards.

The first version of Spamato was a single, monolithic block of code. Of course, the Java *packaging* facility eases the development and maintenance of large projects by bundling related classes in one namespace. But with an increasing number of components in the Spamato system, it was difficult to understand the entire architecture and the dependencies among its individual parts. Moreover, it was almost impossible to integrate new spam filters without breaking the existing code.

The building-block approach offers one chief advantage: a *black box* hides its actual implementation behind a well-defined *interface* facade that defines its functionality. In this chapter, we survey a special type of black boxes: *plug-ins*. More precisely, we examine frameworks that manage a bundle of plug-ins and their interdependencies.

Traditionally, plug-ins have been used to extend the functionality of existing software. For instance, web browsers are extendable to support custom MIME types, such as Java or Flash, and imaging tools can support additional filters. Here, plug-ins are not deployed with the base product but integrated afterwards via a well-defined interface and extension mechanism. We consider a more radical plug-in notion. In our framework, *everything* is a plug-in. Besides a small bootstrapper which initializes and connects

the runtime system, the complete application logic is defined by cooperating plug-ins.

Open-source projects such as the instant messaging client Gaim and the spam filter system SpamPal, which are still traditional frameworks, benefit from third-party developers who contribute additional features, which can be plugged into the existing host application. But only in "everything-is-a-plug-in" frameworks, the full power of plug-in development becomes apparent. For example, Eclipse is not restricted to the default Java IDE—completely unrelated applications can be built on top of the bare framework.

As the development of Eclipse has been driven to provide an application framework for building software, our plug-in framework has been shaped as we have implemented the Spamato spam filter system. However, it proved to be useful in other projects as well.

For Spamato, we expect the development of several third-party spam filters and analyzing tools. Therefore, an architecture that allows the seamless integration of such plug-ins is a must. We also require a mechanism to publish, install, and update plug-ins at runtime, that is, without a restart of the application. This is particularly important for email filtering software because, otherwise, malicious messages can slip through and harm a user's machine. Additionally, we do not want plug-ins to be kept apart, but to interact with each other. For this purpose, we adopt the notion of *hooking* or *extension points* that are associated with well-defined tasks that contributing plug-ins (*extensions*) can implement. Finally, we want a lightweight framework since applications are supposed to run client-side with as little overhead as possible.

In the next section, we discuss existing plug-in frameworks. We detail our own framework in Section 6.2. An example of how the Spamato system makes use of the plug-in mechanism is presented in Section 6.3. Finally, we draw a conclusion in Section 6.4.

6.1 Related Work

The Eclipse framework was initially focused on integrated development environments (IDEs) [109]. With version 3.0, the *Rich Client Platform* (RCP) allows the development of any kind of client applications. The *Eclipse Runtime* comprises the minimum set of classes to build a rich client application. It is built on top of the OSGi framework [127], which defines for instance the life cycle of a plug-in. The orientation on business compatible solutions and the emphasis on *rich* client applications clearly show that Eclipse does not aim to provide a *lightweight* plug-in container but a powerful solution for all circumstances. A minimal "Hello World" project in Eclipse has a footprint of about 450 kB, while our plug-in framework creates about 50 kB only. Our approach adopts some of the features found in Eclipse: mainly the notion of

extensions and extension points as well as the declaration of plug-in interdependencies in a `plugin.xml` descriptor file. But while Eclipse employs a service locator to connect plug-ins, we use the constructor injection pattern to automatically resolve dependencies among plug-ins.

Apache Tomcat is a container that supports the Java Servlet and JavaServer Pages specifications, running J2EE web server applications [147]. In that, it differs from our approach, which rather aims to be embedded in client-side applications. Using Tomcat, dependencies can be modeled by using globally shared directories, which is also reflected in a simple class loader hierarchy. In contrast to Tomcat, which employs a single class loader to enable the sharing of classes among plug-ins, we use a hierarchy of several class loaders to model the interdependencies. Furthermore, Tomcat does not provide any means of extensions or extension points.

The Apache Avalon framework is implemented in several projects, such as Phoenix, Fortress, Merlin, or Codehaus' Loom, see [100] for a description of the project history. We compare our approach to Fortress [112], the only project which provides a lightweight plug-in container. In Fortress, plug-in dependencies are declared as "inline" JavaDoc tags directly in the source code. In contrast, we manage all plug-in dependencies and information in a separate file. Although the Fortress approach seems to entail less overhead, our scheme has the advantage of encapsulating all dependencies in one single file. Moreover, Fortress does not provide extensions and extension points and uses a service locator instead of the constructor injection pattern.

The Codehaus PicoContainer [129] offers a very simple container facility with a footprint of about 50 kB. It provides the constructor and setter injection patterns but has to be configured directly in the source code. No separate descriptor file is needed to model dependencies among plug-ins since all plug-ins run with the same class loader.

A NanoContainer [122] bundles several PicoContainers and allows for the usage of separate class loaders. Additionally, dependencies can be declared in many scripting languages, while we stick to an XML description. There are two main differences to our approach. First, NanoContainer does not implement extensions and extension points. And second, it cannot be altered after the startup—it does not provide install or update features as we use them in our plug-in container.

6.2 The Plug-in Framework

In this section, we describe our plug-in framework [4]. It was originally built to facilitate the development of third-party filters for Spamato, but eases the implementation of any plug-in-based application. Some of our examples are described in the context of Spamato for clearness without loss of generality.

When talking about plug-ins, on the one hand, we mean software components that are *independent blocks* of code. They usually do not provide any features to other components, and do not make use of any shared features. Independent components bundle everything they need to perform a specific task and do not interact with any other component. On the other hand, we think about *open framework blocks* that do not only provide features to other components but explicitly incorporates them, exploiting their capabilities to fulfill a job. It is obvious, that the latter is more powerful and includes the former component type.

In the next section, we formulate the characteristics of our plug-in framework and show how to meet the requirements to support the component types described before. After that, we explain the general process of starting a plug-in-based application and how plug-ins are connected, loaded, and configured. Finally, we highlight the deployment mechanism to publish, install, and update plug-ins.

6.2.1 Plug-In Characteristics

A plug-in features some apparent characteristics such as a name, a description, and a main class. These parameters are generally necessary to manage plug-ins or to provide information to the user. Additional requirements include a security facility restricting the access to local resources, a deployment mechanism to manage different versions of plug-ins, and a scheme to model dependencies between plug-ins.

The `plugin.xml` Descriptor File

Most of the mentioned characteristics are mapped to a `plugin.xml` file that describes a plug-in and its dependencies. The `plugin.xml` descriptor file in Listing 6.1 exemplifies a dummy plug-in.

The aforementioned `<name>` and `<description>` of a plug-in, which are solely of descriptive usage, are listed in lines 2 and 3. The `<class>` denotes a plain old Java class; it does not have to inherit from a "PlugIn" class or implement any interfaces. The `<version>` and `<update-url>` tags provide information for the deployment mechanism, which is detailed in Section 6.2.6.

The `<requires>` section of the XML file specifies security requirements and dependencies on other plug-ins. In this example, the "Dummy PlugIn" requests `"all"` permissions meaning that the plug-in must not be subject to any restrictions enforced by a Java SecurityManager to which permissions are directly mapped. Therefore, this concept enables a fine granular assignment of permissions such as the read/write access to local files from a user directory or the connection to a specific web server only. This is particularly important when dealing with third-party, untrusted plug-ins as described later. Additionally, in the `<requires>` section, a plug-in defines its dependencies

```
 1 <plugin>
 2   <name>Dummy PlugIn</name>
 3   <description>This is a very simple dummy plug-in.</description>
 4   <class>ch.ethz.dcg.dummy.DummyPlugin</class>
 5   <version>1.0</version>
 6   <update-url>http://spamato.net/update</update-url>
 7   <requires>
 8     <permission type="all"/>
 9     <plugin key="another_dummy"/>
10       <extension point="dummy_point" param="hello world"
                    class="ch.ethz.dcg.dummy.DummyExtension"/>
11     </plugin>
12   </requires>
13   <share>
14     <package name="ch.ethz.dcg.dummy.shared"/>
15     <class name="ch.ethz.dcg.dummy.ImportantSharedClass"/>
16     <extension-point id="my_dummy_point"/>
17   </share>
18 </plugin>
```

Listing 6.1: An example of a `plugin.xml` descriptor file.

on other plug-ins—either to get access to `<share>`d classes or resources, or to subscribe to offered "extension points."

The `<share>` part enables other plug-ins to extend or use the facilities provided by the sharing plug-in. In this example, the "Dummy PlugIn" allows other plug-ins to access all classes in the package `ch.ethz.dcg.dummy.shared` and additionally the `ImportantSharedClass`. The sharing of resources, such as images or files, can similarly be achieved. Line 16 states the publishing of an extension point which is further described in the following section.

6.2.2 Extensions and Extension Points

The concept of extending other plug-ins has been borrowed from Eclipse. Plug-ins offer well defined *extension points* which other plug-ins can register with as *extensions*. This approach resembles a publish/subscribe mechanism but is more powerful: Registered extensions are not only notified to handle events, but are expected to extend the capability of the extension point or to perform a particular job.

In Eclipse, for example, many plug-ins add information or views to the user interface by hooking into extension points that are called when the GUI is shown. Thus, new visible elements with associated tasks are embedded into the default editor. The Spamato framework offers several extension points; one is for registering spam filters, which are invoked whenever a new email arrives. In this case, the filtering process is extended or rather relies on what registered extensions contribute.

In Listing 6.1, an `<extension-point>` is defined in the `<share>` section of the plug-in descriptor file (line 16). Its `id` can be referenced in the `<requires>`

part of another plug-in as can be seen in line 10. Besides the `class` that implements the extension point, additional parameters can be set (here "`hello world`"). Usually, the main class of an extension implements an interface which is shared by the plug-in that offers the extension point. In the Spamato system, a plug-in that registers as a spam filter has to implement the `SpamFilter` interface that belongs to the offering plug-in.

6.2.3 Dependency Modeling and Class Loading

Our plug-in mechanism basically represents a lightweight container component which loads plug-ins in a specific format from a specified directory. Plug-ins provide their class files in a `classes` directory (either as individual files or as a single jar file), additional jar files in a `lib` directory, and static resource files, such as documentation files, in an `etc` directory. These directories are located below a `bin` directory which also holds the `plugin.xml` file. Further dynamic content that is created at runtime, such as user defined configuration files, are stored in the root directory of each plug-in.

All plug-ins are located in the *profile directory* which is recursively traversed when a plug-in-based application is started. As mentioned earlier, a `plugin.xml` file characterizes the interaction (required plug-ins for shared classes and extension points) with other plug-ins. These files are parsed in order to build a directed, acyclic graph which reflects all dependencies of all plug-ins. The graph is used, for example, to determine the start-up order, such that plug-ins are available when dependent ones need them. Note that the container is partly implemented as a plug-in itself (the *runtime* plug-in). It shares classes and resources, offers extension points, and exhibits all other features of a normal plug-in. Thus, the runtime plug-in is also contained in the graph but does not depend on any other plug-in.

This graph is also reflected in a similar hierarchically organized set of Java `ClassLoaders`. Generally, each plug-in is managed by its dedicated class loader. By default, no plug-in can use or even knows about other plug-ins; they are totally shielded in their personal namespaces. This results in four nice features. First, developers do not have to worry about other plug-ins. They can label their packages without considering problems due to any name collisions even though all plug-ins are dynamically loaded into the same JVM. More precisely, developers can prohibit access to their classes. Second, we can easily assign different individual security permissions as mentioned earlier. Third, the testing and analysis of plug-ins with different parameters is eased. In the Spamato system for instance, spam filters, which are plug-ins themselves, can be used multiple times in one Spamato instance with different settings by copying them into different directories in the profile directory. Thus, it is possible to easily compare the results of the same filters running at the same time with different settings. Finally, using separate class loaders provides the capabilities to update plug-ins without the need for a

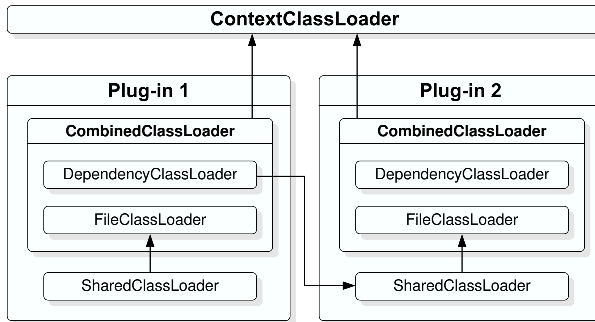

Figure 6.1: The class loader hierarchy of two plug-ins.

restart of the whole container component. Instead, only the updating plug-in and its dependent plug-ins need to be restarted which can be performed at runtime.

Yet we still have to provide the sharing of classes, resources, and extension points. This means, we need some facility to let other plug-ins make use of the shared information. Additionally, hooking into an extension point entails that the plug-in which offers the extension point calls methods of the extending plug-ins—mutually connecting both plug-ins with each other.

To allow such interactions, the default Java class loading scheme has to be adapted. In our plug-in system, each plug-in is backed by four different types of class loaders: the *FileClassLoader*, the *SharedClassLoader*, the *DependencyClassLoader*, and the *CombinedClassLoader*. The interaction of two plug-ins and their class loaders is depicted in Figure 6.1.

The *FileClassLoader* is responsible to load files from the file system and is restricted to the associated plug-in directory. The *SharedClassLoader* wraps the *FileClassLoader*. It restricts the access for other plug-ins to those files which are declared as "shared" in the plug-in descriptor file. The *DependencyClassLoader* enables plug-ins to access shared classes of other plug-ins by using their *SharedClassLoader*, provided that a dependency is declared. Finally, the *CombinedClassLoader* combines the *FileClassLoader* and the *DependencyClassLoader* and provides all class files, resources, and libraries accessible through them to the associated plug-in. Furthermore, there is a single *ContextClassLoader*. It enables all *CombinedClassLoaders* to access the default Java classes, the bootstrap plug-in classes, which are not part of any plug-in, and all other classes and libraries that can be found in the default `CLASSPATH`.

Also note that our class loading mechanism differs from the default Java "first parent/then child"-scheme. We first browse the directly associated

FileClassLoader, and only if the class or resource can not be accessed, the parent CombinedClassLoader is called.

6.2.4 The Life Cycle of a Plug-in

The life cycle of a plug-in in our framework is simple. It can be managed by implementing specific interfaces instead of hooking into an extension point. The plug-in descriptor file is loaded and parsed in the initialization phase by a *Plugin Handler*; for each plug-in exists one handler. After all plug-ins have been initialized, they are loaded respecting the order of the directed acyclic graph described in Section 6.2.3. Thus, whenever a plug-in is loaded, its required plug-ins are available. The loading and instantiation of classes is performed using the *Dependency Injection* pattern (IoC) or more precisely, the constructor-based injection variant of it. References to required plug-ins are automatically assigned through constructor parameters which are resolved using the Java *reflection* facility.[1] The *start* and *dispose* phases of the life cycle can optionally be handled by implementing the corresponding interfaces.

6.2.5 Configuration

An interface unifies the access to configuration settings from various sources. The individual settings object of a plug-in can be accessed as a constructor parameter during the start-up phase as described in the previous section.

Currently, text, Java properties, and XML files are supported to store the settings; additional formats can be contributed using the corresponding extension point of the runtime plug-in. For instance, a database implementation would be more appropriate to support a large number of users for a server-side Spamato version.

6.2.6 The Deployment Mechanism

Oreizy et al. [71] identify three types of architectural changes in the life-time of plug-ins in a framework: the *addition* of plug-ins, the *removal* of plug-ins, and the *replacement* of plug-ins. We refer to these types as the *installation*, *deletion*, and *update* of plug-ins, respectively. Furthermore, we extend our framework to allow for another aspect: the *publication* of new plug-ins by any user.

The runtime plug-in provides an extension point for each of these four aspects; this is illustrated in Listing 6.2. Plug-ins can contribute to these extension points by implementing corresponding interfaces.

[1] We also provide the service locator approach by allowing plug-ins to access the plug-in container. But we regard the constructor injection method to be easier to maintain—and a reference to the service locator can only be accessed in this way.

```
<plugin>
  <name>Runtime</name>
  <share>
    <extension-point id="deploy.search"/>
    <extension-point id="deploy.download"/>
    <extension-point id="deploy.upload"/>
    <extension-point id="deploy.publish"/>
  </share>
  <extension point="deploy.search" id="http" class="..."/>
  <extension point="deploy.download" id="http,ftp,file" class="..."/>
</plugin>
```

Listing 6.2: The runtime plug-in offers extension points to provide deployment handlers and registers default ones.

Search handlers are used to find plug-ins that can be installed or updated. We provide default search handlers for HTTP and FTP servers as well as for the local file system. The search provides a list that contains fragments of the `plugin.xml` of a plug-in: descriptive data, such as the name, as well as data necessary for the update mechanism, for instance the version and the download source. *Download* handlers fetch plug-ins from a download server. As an additional plug-in, the tracker-based Peerato system allows downloading files directly from other users using a proprietary protocol. *Upload* handlers store plug-ins on a download server, and *publish* handlers update the list of available plug-ins accessed by search handlers. Both are necessary in order to make new or updated plug-ins available to other users.

Note that the runtime plug-in only provides the basic capability to manage the deployment cycle of plug-ins. To employ it in an accessible way, further steps have to be taken. In the Spamato system, we use a browser-based approach to cope with this issue.

The Profile Deployment Scheme

On multi-user platforms, such as Linux and (to some extent) Windows XP, it is often feasible to install an application based on our framework for all users only once. This eases the application maintenance, for example when updating plug-ins. Still, users should be able to configure their personal environment, for instance by installing custom plug-ins or removing default ones.

We address this issue in our *profile deployment scheme*. Usually, an administrator installs an application to a directory which users can only read from, the *default* installation directory. To allow for the configuration of a user-specific environment, plug-ins are installed to the user's *profile* directory when an application is started for the first time. Any modification can now be performed in the profile directory instead of the read-only installation directory. Nevertheless, administrators can update the default installation and our profile deployment scheme applies the changes to each user profile.

```
<plugin>
  <name>Current Filter Process</name>
  <share>
    <extension-point id="precheckers"/>
    <extension-point id="filters"/>
    <extension-point id="decision_makers"/>
    <extension-point id="postcheckers"/>
    <package name="ch.ethz.dcg.cufip">
  </share>
</plugin>
```

Listing 6.3: The CuFiP offers several extension points to control the process of filtering emails.

6.3 Using the Plug-In Framework in the Spamato System

In the previous chapters, we described a variety of plug-ins that exist in the Spamato system. The obvious key functionality of Spamato is to check whether incoming emails are spam. This task is performed by several spam filter plug-ins, which are managed by a filter process component as described in Section 2.2. As an example, we briefly show how the filter process is defined in the plug-in framework. Thereafter, we sketch the implementation of the Earlgrey filter, which depends on several other plug-ins.

The Filter Process

The `plugin.xml` file that exposes the capabilities of the CuFiP (see Section 2.2.2) to the plug-in framework is given in Listing 6.3. The defined extension points directly correspond to the different filter phases. For instance, post-checkers have to register with the "`postcheckers`" extension point in order to be notified after the classification of an email has been determined. Moreover, for each extension point exists an associated Java interface that has to be implemented by a plug-in for compatibility reasons. That is, all post-checkers have to implement the "`PostChecker`" interface that contains the invoked "`onPostCheck()`" method. This interface is defined in the "`ch.ethz.dcg.cufip`" package, which is shared with other plug-ins.

The Earlgrey Filter

The `plugin.xml` of the Earlgrey filter is shown in Listing 6.4. As described in Section 3.6, the Trooth system is used to weight votes from other users with their trust values. The *url* plug-in extracts the relevant domains contained in an email, and previously cached results can be consulted using the *filter history* component. The *runtime* plug-in provides basic utility functions to all plug-ins that want to store configuration files. Finally, the settings of a plug-in can be managed using the *web config* plug-in; for the Earlgrey filter, two pages are registered.

```
<plugin>
  <name>Earlgrey Filter</name>
  <requires>
    <plugin key="cufip">
      <extension point="filters" class="..."/>
    </plugin>
    <plugin key="trooth"/>
    <plugin key="url"/>
    <plugin key="filterhistory"/>
    <plugin key="runtime"/>
    <plugin key="webconfig" name="Configuration">
      <extension point="config.pages" class="..."
                 page="" menu="Earlgrey"/>
      <extension point="config.pages" class="..."
                 page="aggressiveness" menu="Aggressiveness"/>
    </plugin>
  </requires>
</plugin>
```

Listing 6.4: The Earlgrey filter depends on several other plug-ins to perform its filtering task.

6.4 Concluding Remarks

In this chapter, we presented a lightweight but powerful plug-in container, which provides advanced features such as dynamic class loading and dependency, configuration, and security management. The plug-in container is an important part of the Spamato system, as it eases the integration of third-party spam filters.

The plug-in framework was designed to support single-user applications. A multi-user installation—one that shares a common code-base, runs in a single JVM, and differs only in the configuration files for each user—is currently not supported.[2] However, we plan to run the Spamatoxy in such a scenario: A single instance could serve many users in parallel. For multi-user support, we plan to extend the plug-in framework such that it uses *session* objects to distinguish between users. Additionally, it might be necessary to provide further tools to ease the administration of users and plug-ins.

[2]It is possible to start the same application several times in a multi-user environment using independent Java processes. However, running it only once in a single JVM for distinct users is not supported.

Chapter 7

Project Statistics and Filter Results

`My girlfriend loves the results, but she doesn't know what I do.`
`She thinks it's natural`
`(zona armstrong <dalemae@my.justlove.kiev.ua>, 6/3/2006)`

In this chapter, we present some statistics about Spamato which we collected during the last few weeks. The data was mostly taken from our statistics database (see Section 2.4.2), which contains detailed information about detected spam messages, such as the involved spam filters and their parameters. The database is an extensive source for the real-world analysis of the deployed system. However, its maintenance is also "expensive," as for each detected spam message a notification is sent from the users and a record has to be stored on the server. Nevertheless, the insights gained from this data provide valuable information about the entire Spamato system running in hundreds of different environments and can help improve it in the future.

In the next section, we give a brief overview of the project history. Section 7.2 presents statistics about the development of the number of users and detected spam messages. Thereafter, in Section 7.3 we take a closer look at the performance of the Spamato filters. Finally, in Section 7.4 we draw a brief conclusion.

7.1 Project History

The first version of Spamato was published in February 2004. We started using SourceForge with version 0.9 in August 2005, followed by version 0.98 in December 2005 and version 0.98b in February 2006. Version 0.99 was released on June, 11th, 2006 on SourceForge and was added to the add-ons directory of Mozilla on July, 20th, 2006. In the following sections, we considered only version 0.99 for which all data was measured until August, 7th, 2006.

The latest Spamato release was downloaded about 4500 times from the SourceForge and Mozilla sites: Spamato4Thunderbird had about 2750 downloads, Spamato4Outlook 1160, Spamatoxy 510, and Spamatozilla 80. Spamato4Outlook was most frequently downloaded until Spamato4Thunderbird was also available from the Mozilla page; more than 1600 downloads were registered for this extension during the last 3 weeks.

Spamato was several times among the 100 most active projects on SourceForge; its best rank was 58 on August, 4th. The activity of a project is based on several criteria, such as the number of downloads, homepage visits, bug and feature tracker entries, and forum messages. Spamato had to compete with more than 100000 other projects which were hosted on SourceForge as well.

Searching for the term "Spamato" on Google produced 17300 hits on August, 7th (not all for our software, though). It was added to a variety of download platforms, discussed in blogs, and recommended in several articles. Each day, hundreds of people visit our homepage at `http://www.spamato.net` browsing for information and statistics. We expect to get further attention with version 1.0 of Spamato.

7.2 Usage Statistics

Although Spamato was downloaded about 4500 times, the number of active users was (for unknown reasons) smaller: For only 1713 users did our statistics database record at least one successfully detected spam message. 1332 of these users were active on at least two days, and 569 users on at least seven days. Currently, about 50 new users join the Spamato community every day.

Figure 7.1 illustrates the development of new and active users over time, starting on June, 6th, as day "1".[1] The number of *known users* is the cumulative sum of new users. The ratio between active and known users per day is mapped on the second y-axis and declines over time. Nevertheless, the total number of active users shows a clear upward trend, and we expect the ratio between active and known users to stabilize in the future. Note the drastic increase in the number of users since day "44" when Spamato4Thunderbird was listed on the Mozilla directory for the first time.

In total, Spamato classified 537544 messages as spam. Currently, the number of daily spam detections is larger than 30000. More than 100 messages were removed by Spamato from 593 users' inboxes, 107 users received more than 1000 spam messages, and Spamato helped the worst affected user in 24303 cases. Figure 7.2 depicts the number of daily detected spam messages. As before, since day "44" we can see a significant increase in the number of identified spam emails. Interestingly, the average number of spam emails per user, as depicted on the second y-axis, almost stabilized over time and now slowly increases to about 60 spam messages per user and day.

[1] Note that we had an internal release five days before we published it on SourceForge.

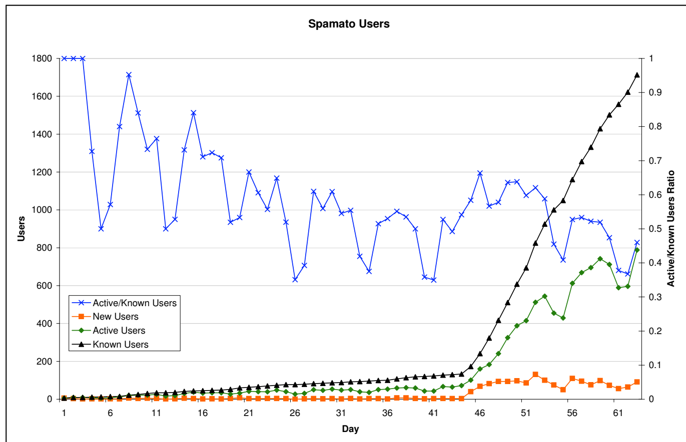

Figure 7.1: This figure depicts the number of new and active users per day since the first release day of version 0.99. The considerable increase since day "44" results from the release of Spamato on the Mozilla directory in addition to SourceForge.

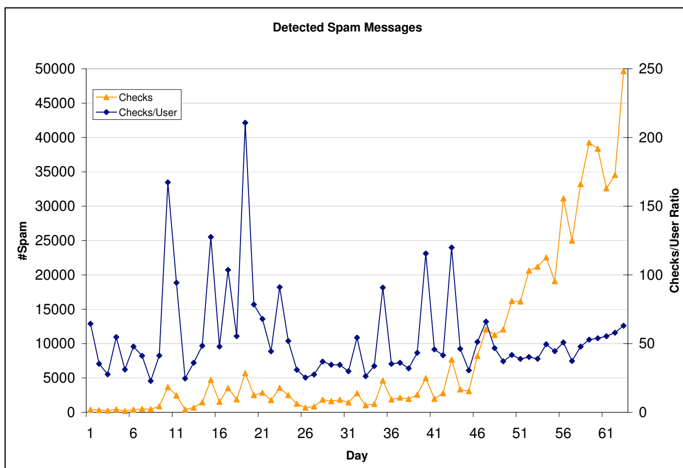

Figure 7.2: This figure illustrates the number of detected spam messages per day and per user. Again, the increase since day "44" is due to the listing of Spamato4Thunderbird on the Mozilla site.

7.3 Filter Results

Overall, our statistics database recorded 537544 spam detections from which 13053 have been revoked (about 2.4%). Most users kept the default settings for the filters and the *Min-Spam Decision Maker* (see Section 2.3.1); only few decided to modify the parameters. For simplicity, we did not consider different parameter values; doing so would lead to slightly different results though.

Note that we did not collect information about detected ham messages. We can elaborate only on the number of spam detections and the number of reported and revoked emails. To complicate the matter, reports did not necessarily reflect false negatives since messages were also reported when training the Bayesianato. That is, we cannot distinguish between real false negatives and "training" reports. Similarly, not all revokes were false positives. In this case, we determined the number of real false positives as the number of revoked messages for which also a positive spam check was stored in the database.

In the following paragraphs, we present the results concerning spam detections and false positive rates and discuss the impact of different filter selections and min-spam values.

Detection Rate

Figure 7.3 illustrates how many spam emails were detected by each filter; the "Spamato" category reflects the entire Spamato system with all filters. The Razor.Ephemeral and Razor.Whiplash engines were part of the Razor filter and not individually evaluated. We show them here, as this allows us to compare the Whiplash algorithm with the Earlgrey filter, which are both collaborative and URL-based filters. Furthermore, the Ruleminator is not listed, as all of its rules were evaluated but not the Ruleminator on its own. The Rule.* rules are default rules in the Ruleminator and were employed by most users. Some users also created their own rules. However, we decided to discard them from this overview, as they had only a very small impact on the overall results.

Note that none of the filters checked all emails. For instance, the Bayesianato checked only 311000 messages—226000 were not checked, as the Bayesianato was either still "learning"—that is, it had not seen enough tokens to start classifying emails—or had been disabled by the user. The Earlgrey filter checked only about 432000 emails—the rest was not classified, as those emails did not contain any domain (or because the Earlgrey filter had been disabled).

The figure also shows the detection rate for each filter. For instance, the Bayesianato detected about 88% of all spam emails it checked whereas the Domainator classified only 45% of its checked message as spam; this

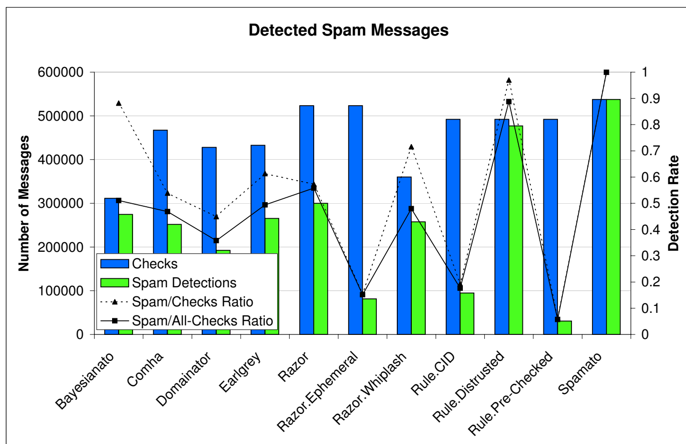

Figure 7.3: This figure shows the number of checked and detected spam messages for each filter. The *spam/checks ratio* is calculated by dividing the number of detected spam messages by the number of checked messages per filter. For the *spam/all-checks ratio*, the number of spam detections is divided by the number of *all* (Spamato) checks.

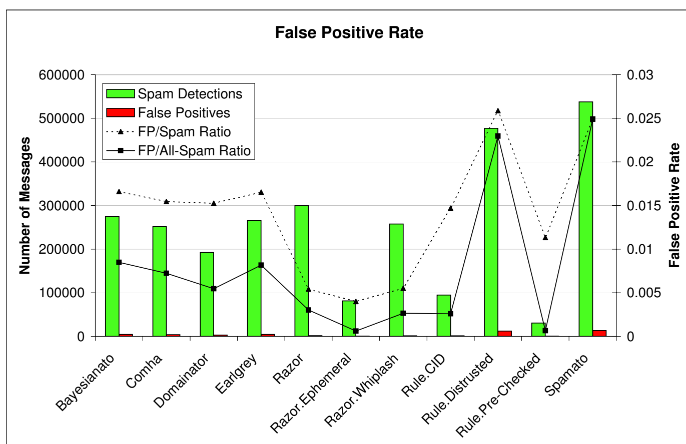

Figure 7.4: This figure illustrates the false positive rate of each filter. The *fp/spam ratio* is calculated as the ratio between false positives and detected spams per filter. Similarly, the *fp/spam-all ratio* is the number of false positives divided by the number of *all* spam messages.

spam/checks ratio is depicted as the dashed line with triangle markers. The second curve marked with squares is the ratio of the number of spam detections per filter divided by the number of *all* checks (*spam/all-checks ratio*). For instance, the Earlgrey filter had a *spam/checks ratio* of 61% and a *spam/all-checks ratio* of only 49% since it did not check all emails as mentioned above.

False Positive Rate

The false positive rate for each filter is illustrated in Figure 7.4. It was calculated as the number of false positives (revokes) in which a filter was involved divided by the number of spam detections per filter (*fp/spam ratio*) and by the total number of spam detections (*fp/all-spam ratio*). The minimal number of 2 filters per false positive was given in 9553 cases, 3 filters were responsible for 2389 misclassifications, and 4 or even more filters erroneously voted in 1111 cases for spam. Therefore, the false positive rates of the individual filters cannot be summed up to meet the overall rate, as the sets of misclassified emails were not distinct.

The overall false positive rate was about 2.4% as given in the "Spamato" column. The fp/spam ratio for the Bayesianato, for example, was about 1.7% and the fp/spam-all ratio was approximately 0.8%. As can be seen, the Razor filter performed best; it is the only filter with an fp/spam ratio below 1%. The worst performing filter was the Rule.Distrusted rule. It was involved in about 2.6% of all misclassifications, which was not surprising since this rule classifies on a very strict criterion.

Impact of Min-Spam Values

In this section, we briefly discuss the impact of choosing different min-spam values on the filtering success. Figure 7.5 depicts the false positive and false negative rates for the entire Spamato system. The false positive rate is measured as before; for instance, for a min-spam value of 2, it is about 2.4%. The false negative rate is calculated as 1 minus the number of missed spam messages divided by the number of all spam messages. For example, with a min-spam value of 3, about 133000 emails (24%) would not be detected as spam anymore. At the same time, the false positive rate would also be decreased to only 0.9%.

Obviously, choosing high min-spam values results in low detection rates. Therefore, we chose 2 as the default value—accepting a slightly higher false positive rate than with other values.

Impact of Filter Selection

In this experiment, we were interested in the contribution that each filter had on the overall detection rate. Particularly, we measured the false positive and false negative rates of the entire Spamato system when one filter was removed. The results for a min-spam value of 2 are illustrated in Figure 7.6.

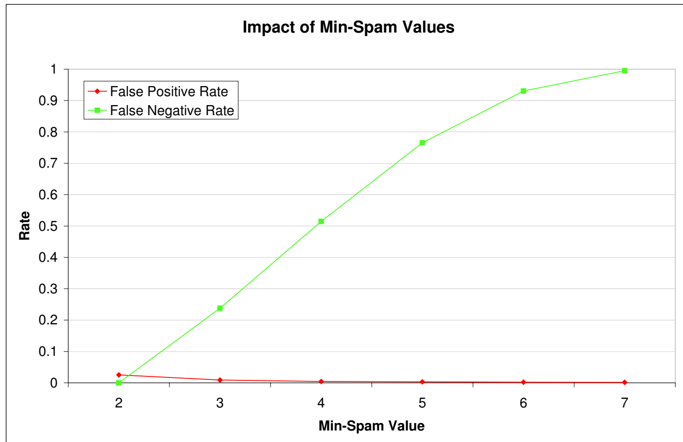

Figure 7.5: This figure depicts the trade-off between the false positive and false negative rates for different min-spam values. For example, choosing a min-spam value of 3 instead of 2 will decrease the detection rate by about 24% while the false positive rate will be reduced by about 1.5%.

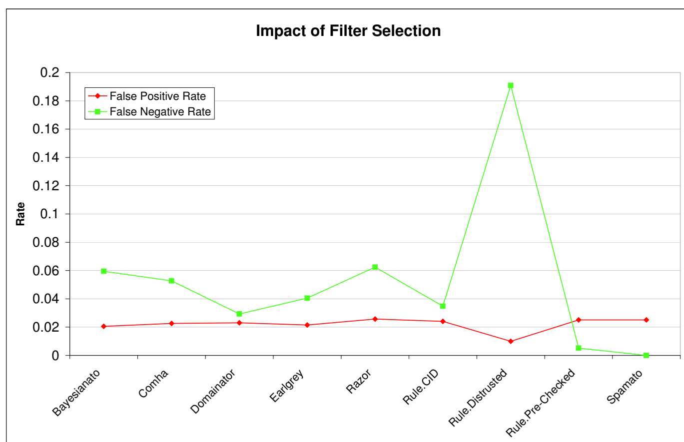

Figure 7.6: The figure depicts the impact on the false positive and false negative rates when one filter is removed. For instance, discarding the Bayesianato from the set of Spamato filters decreases the detection rate by about 6% while the false positive rate is reduced from 2.4% to 2%.

For example, when removing the Bayesianato, the remaining seven filters would have a false positive rate of 2% and a false negative rate of 6%. That is, only about 505000 emails were detected as spam, and thereof about 10500 were false positives. In other words, 32000 spam messages were not detected if we had discarded the Bayesianato from the Spamato system since only one other filter detected them as well. Note that removing the Rule.Distrusted filter would have the highest impact on the false positive and false negative rates. Again, this is because of its very harsh classification scheme.

The main purpose of this figure is to show that all filters contribute to the overall detection rate. Discarding any individual filter would reduce the number of detected spam messages. Moreover, the false positive rate decreases only slightly when removing a filter. In conclusion, utilizing all eight filters in the Spamato system seems to be most advantageous compared to any "seven-filter system."

We also measured the effectiveness of all 28 two-filter combinations. The best performing combination (Bayesianato and Rule.Distrusted) detected only about 52% of all spam messages with a false positive rate of 0.8%; Razor in combination with the Rule.Distrusted filter reached a detection rate of 50% with a 0.3% false positive rate. These results confirm that bundling several filters in a spam filter system is a good choice.

7.4 Concluding Remarks

The results presented in this chapter were derived from our statistics database, which proved to be an invaluable source of information. It was interesting to analyze a live system containing hundreds of users with different usage behaviors. However, this variety also had some drawbacks. For instance, the results indicated that Spamato performed worse than many spam filters evaluated in the literature. One reason for this behavior could be that many new users joined Spamato in the evaluated period. Since new users generally start with untrained filters, it is possible that false positives are more common until the filters get more training data. Another explanation could be that spam filters in general do not meet the promised performance for the "average real-world user." It has to be studied further if the Spamato filters are really that poor.

Tom Fawcett argued in [28] that "real-world *in vivo* spam filtering is a rich and challenging problem for data mining. By 'in vivo' we mean the problem as it is truly faced in an operating environment, that is, by an on-line filter on a mail account that receives realistic feeds of email over time, and serves a human user." In this chapter, we actually did not study "in vivo" *spam filtering*, but looked at a related domain: "in vivo" *spam filtering results*. Moreover, we studied this domain not for a single but for hundreds of "human users." We believe that our data will help gain a better understanding of spam filtering in a real-world scenario.

Chapter 8

Conclusion

> `I'm just a regular person who has seen what the world has to offer,`
> `thus I have come to the conclusion I love all the people...`
> `(Violet Guerrero <atthistveg@atthis.com>, 6/8/2006)`

A crowd of drunken Vikings bawl Monty Python's *Spam Song* whenever Spamato detects a spam message. "Spam spam spam spam. Lovely spam! Wonderful spam!" are the words that conclude the *mail-way to hell*. The song is played at the very end of the filter process, in which the true face of every email is revealed. The filter process is the conductor of the Spamato system. It directs the invocation order of pre-checkers, spam filters, and decision makers when scrutinizing emails. We discussed several implementations of these components and their interaction, as they are crucial for the overall success of Spamato.

The work presented in this thesis has a strong practical background. Spamato was implemented for the purpose of being employed by many users in real-world scenarios. This was a challenging venture, as reality significantly differs from test bed experiments. Nonetheless, we succeeded in achieving this goal: Hundreds of people use Spamato every day to remove unsolicited emails from their inboxes. Since all filter results are stored in our statistics database, we are able to analyze how each filter—and the entire Spamato system—performs in operating environments.

We designed Spamato to allow for the integration of third-party spam filters. With the extension mechanism, it is easy to plug additional filters into the system. Unfortunately, we have not yet been able to encourage any spam filter developer to rely on the functionality provided by the Spamato framework. A comprehensive "Developer's Guide" may help to attract programmers who want to contribute to the Spamato project. We also plan to implement new spam filters on our own. Particularly, we want to explore the capabilities of collaborative spam filters in more depth.

The long-term goal of our project is to turn Spamato into a kind of "smart secretary" for email handling. Currently, Spamato is restricted to the

spam filtering domain; but this restriction is self-imposed. From a technical perspective, we can easily support mechanisms to make the handling of email in general more feasible. Many people complain about not being able to keep up with the load of emails arriving in their inboxes every day—and this concerns *ham* messages! Once a message has scrolled "off the screen," it is most likely lost forever. We recently started research on automatic topic detection and prioritization of emails. Employing such mechanisms will help organize what is left after removing unsolicited content, urging users to answer the most important messages first.

At the end of this thesis, we want to recall its very beginning: "Mastering Spam" was the challenge of this dissertation. Did we succeed? *No* would be an honest answer—when measured on a global scale, we achieved only little. Spam is still a problem, and it seems to get even worse. However, the objective of this project was not to eradicate spam completely. In this thesis, we sought to *filter* spam upon reception. We did not fight the root of the problem, which would prevent spammers from sending spam at all. So maybe, *yes* is the correct answer. At least, the existence of hundreds of people successfully using Spamato provides evidence that this is indeed the case. Who should decide about this question if not the users?

At the World Economic Forum in January 2004, Bill Gates claimed that "Two years from now, spam will be solved." Looking at the current situation, he was undoubtedly wrong. Comparing Spamato to Microsoft's efforts may appear audacious. But only time will tell to which extent Spamato can master spam.

Bibliography

[1] K. Albrecht, R. Arnold, M. Gähwiler, and R. Wattenhofer. Aggregating Information in Peer-to-Peer System for Improved Join and Leave. In *Proceedings of the International Conference on Peer-to-Peer Computing (P2P)*, 2004.

[2] K. Albrecht, R. Arnold, and R. Wattenhofer. Clippee: A Large-Scale Client/Peer System. In *Proceedings of the International Workshop on Large-Scale Group Communication, held in conjunction with the 22nd Symposium on Reliable Distributed Systems (SRDS)*, 2003.

[3] K. Albrecht, N. Burri, and R. Wattenhofer. Spamato - An Extendable Spam Filter System. In *Proceedings of the Second Conference on E-mail and Anti-Spam (CEAS)*, 2005.

[4] K. Albrecht and R. Wattenhofer. Development, Deployment, and Rating of Plug-Ins. Technical Report TIK-259, Computer Engineering and Networks Laboratory, ETH Zurich, August 2006.

[5] K. Albrecht and R. Wattenhofer. The Trooth Recommendation System. In *Proceedings of the International Conference on Internet and Web Applications and Services (ICIW)*, 2006.

[6] E. Allman, J. Callas, M. Delany, M. Libbey, J. Fenton, and M. Thomas. Domainkeys Identified Mail (DKIM). Internet Engineering Task Force (IETF) Draft, 2005.

[7] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, G. Paliouras, and C. D. Spyropoulos. An Evaluation of Naive Bayesian Anti-Spam Filtering. In *Proceedings of the Workshop on Machine Learning in the New Information Age in conjunction with the European Conference on Machine Learning (ECML)*, 2000.

[8] I. Androutsopoulos and Paliouras. Learning to filter unsolicited commercial E-Mail. Technical Report TR 2004/2, National Centre for Scientific Research, "Demokritos", March 2004.

[9] B. Awerbuch, Y. Azar, Z. Lotker, B. Patt-Shamir, and M. R. Tuttle. Collaborate With Strangers To Find Own Preferences. In *Proceedings of 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2005.

[10] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Tuttle. Improved Recommendation Systems. In *Proceedings of 16th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005.

[11] J. Aycock and N. Friess. Spam Zombies from Outer Space. Technical Report TR 2006-808-01, University of Calgary, Canada, January 2006.

[12] O. Bälter and C. L. Sidner. Bifrost Inbox Organizer: Giving users control over the inbox. In *Proceedings of the Second Nordic Conference on Human-Computer Interaction (NordiCHI)*, 2002.

[13] I. Bhattacharya, S. R. Kashyap, and S. Parthasarathy. Similarity Searching in Peer-to-Peer Databases. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS)*, 2005.

[14] P. O. Boykin and V. P. Roychowdhury. Leveraging Social Networks to Fight Spam. *Computer*, 38(4):61–68, 2005.

[15] S. Brin, J. Davis, and H. Garcia-Molina. Copy Detection Mechanisms for Digital Documents. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1995.

[16] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the Web. *Computer Networks and ISDN Systems*, 29(8-13):1157–1166, 1997.

[17] B. Burton. SpamProbe - Bayesian Spam Filtering Tweaks. In *Proceedings of the Spam Conference*, 2003.
`http://spamprobe.sourceforge.net/paper.html`.

[18] A. Chowdhury, O. Frieder, D. Grossman, and M. C. McCabe. Collection Statistics for Fast Duplicate Document Detection. *ACM Transactions on Information Systems*, 20(2):171–191, 2002.

[19] W. W. Cohen. Learning Rules that Classify E-Mail. In *Proceedings of the AAAI Spring Symposium on Machine Learning in Information Access*, 1996.

[20] J. G. Conrad, X. S. Guo, and C. P. Schriber. Online Duplicate Document Detection: Signature Reliability in a Dynamic Retrieval Environment. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 2003.

[21] G. V. Cormack and A. Bratko. Batch and On-line Spam Filter Comparison. In *Proceedings of the Third Conference on Email and Anti-Spam (CEAS)*, 2006.

[22] G. V. Cormack and T. R. Lynam. TREC 2005 Spam Track Overview. In *Proceedings of the Text REtrieval Conference (TREC)*, 2005.

[23] G. Cselle, K. Albrecht, and R. Wattenhofer. BuzzTrack: Topic Detection and Tracking in Email. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, 2007.

[24] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. An Open Digest-based Technique for Spam Detection. In *Proceedings of the International Workshop on Security in Parallel and Distributed Systems (PDCS)*, 2004.

[25] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. P2P-Based Collaborative Spam Detection and Filtering. In *Proceedings of the Fourth International Conference on Peer-to-Peer Computing (P2P)*, 2004.

[26] T. V. Dinter. New and Upcoming Features in SpamAssassin v3. In *Talk at the ApacheCon US*, 2004.

[27] P. Drineas, I. Kerenidis, and P. Raghavan. Competitive Recommendation Systems. In *Proceedings of 34th ACM Symposium on Theory of Computing (STOC)*, 2002.

[28] T. Fawcett. “In vivo” spam filtering: A challenge problem for KDD. *KDD Explorations*, 5(2):140–148, 2003.

[29] T. Fawcett. ROC Graphs: Notes and Practical Considerations for Data Mining Researchers. Technical Report HPL-2003-4, Hewlett Packard Labs, April 2003.

[30] R. A. Finkel, A. Zaslavsky, K. Monostori, and H. Schmidt. Signature extraction for overlap detection in documents. In *Proceedings of the Australasian Computer Science Conference (ACSC)*, 2002.

[31] G. Forman, K. Eshghi, and S. Chiocchetti. Finding Similar Files in Large Document Repositories. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, 2005.

[32] A. Garg, R. Battiti, and R. Cascella. “May I Borrow Your Filter?” Exchanging Filters to Combat Spam in a Community. Technical Report DIT-05-089, University of Trento, Italy, November 2005.

[33] S. Garriss, M. Kaminsky, M. J. Freedman, B. Karp, D. Mazieres, and H. Yu. RE: Reliable Email. In *Proceedings of the Symposium on Networked Systems Design & Implementation (NSDI)*, 2006.

[34] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using Collaborative Filtering to Wave an Information Tapestry. *Communications of the ACM*, 35(12):51–60, 1992.

[35] D. Goodman. *Spam Wars*. SelectBooks, 2004.

[36] J. Graham-Cumming. The Spammers' Compendium. In *Proceedings of the Spam Conference*, 2003. `http://popfile.sourceforge.net/SpamConference011703.pdf`.

[37] J. Graham-Cumming. How to Beat a Bayesian Spam Filter. In *Proceedings of the Spam Conference*, 2004.

[38] A. Gray and M. Haahr. Personalised, Collaborative Spam Filtering. In *Proceedings of the First Conference on E-mail and Anti-Spam (CEAS)*, 2004.

[39] Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen. Combating Web Spam with TrustRank. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[40] J. Haitsma, T. Kalker, and J. Oostveen. Robust Audio Hashing for Content Identification. In *Proceedings of International Workshop on Content-Based Multimedia Indexing (CBMI)*, 2001.

[41] S. Han, Y. Ahn, S. Moon, and H. Jeong. Collaborative Blog Spam Filtering Using Adaptive Percolation Search. In *Proceedings of the Workshop on the Weblogging Ecosystem*, 2006.

[42] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl. An Algorithmic Framework for Performing Collaborative Filtering. In *Proceedings of the 1999 Conference of the American Association of Artificial Intelligence (AAAI)*, 1999.

[43] S. Hershkop. *Behavior-based Email Analysis with Application to Spam Detection*. PhD thesis, Columbia University, 2006.

[44] S. Hershkop and S. J. Stolfo. Combining Email Models for False Positive Reduction. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, 2005.

[45] G. Hulten, A. Penta, G. Seshadrinathan, and M. Mishra. Trends in Spam Products and Methods. In *Proceedings of the First Conference on E-mail and Anti-Spam (CEAS)*, 2004.

[46] J. Jung and E. Sit. An Empirical Study of Spam Traffic and the Use of DNS Black Lists. In *Proceedings of the Internet Measurement Conference (IMC)*, 2004.

[47] J. J. Jung and G.-S. Jo. Collaborative Junk E-mail Filtering Based on Multi-agent Systems. In *Proceedings of the Second International Conference on Human Society@Internet (HSI)*, 2003.

[48] B. Kerr and E. Wilcox. Designing Remail: Reinventing the Email Client Through Innovation and Integration. In *Proceedings of the Conference On Human Factors In Computing Systems (CHI)*, 2004.

[49] J. Kittler, M. Hatef, R. P. W. Duin, and J. Matas. On Combining Classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(3):226–239, 1998.

[50] J. Kleinberg and M. Sandler. Convergent Algorithms for Collaborative Filtering. In *Proceedings of ACM Conference on Electronic Commerce (EC)*, 2003.

[51] B. Klimt and Y. Yang. The Enron Corpus: A New Dataset for Email Classification Research. In *Proceedings of the European Conference on Machine Learning (ECML)*, 2004.

[52] A. Kołcz, A. Chowdhury, and J. Alspector. The Impact of Feature Selection on Signature-Driven Spam Detection. In *Proceedings of the First Conference on E-mail and Anti-Spam (CEAS)*, 2004.

[53] J. S. Kong, P. O. Boykin, B. A. Rezaei, N. Sarshar, and V. P. Roychowdhury. Scalable and Reliable Collaborative Spam Filters: Harnessing the Global Social Email Networks. In *Proceedings of the Second Conference on E-mail and Anti-Spam (CEAS)*, 2005.

[54] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Recommendation systems: a probabilistic analysis. In *Proceedings of 39th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998.

[55] H. Lee and A. Y. Ng. Spam Deobfuscation using a Hidden Markov Model. In *Proceedings of the First Conference on E-mail and Anti-Spam (CEAS)*, 2004.

[56] B. Leiba, J. Ossher, V. T. Rajan, R. Segal, and M. Wegman. SMTP Path Analysis. In *Proceedings of the Second Conference on E-mail and Anti-Spam (CEAS)*, 2005.

[57] J. R. Levine. Experiences with Greylisting. In *Proceedings of the Second Conference on E-mail and Anti-Spam (CEAS)*, 2005.

[58] S. Lin, M. T. Özsu, V. Oria, and R. Ng. An Extendible Hash for Multi-Precision Similarity Querying of Image Databases. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, 2001.

[59] G. Linden, B. Smith, and J. York. Amazon.com Recommendations, Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[60] D. Lowd and C. Meek. Adversarial Learning. In *Proceedings of the International Conference on Knowledge Discovery in Data Mining (KDD)*, 2005.

[61] D. Lowd and C. Meek. Good Word Attacks on Statistical Spam Filters. In *Proceedings of the Second Conference on Email and Anti-Spam (CEAS)*, 2005.

[62] J. Lyon and M. Wong. Sender ID: Authenticating E-Mail, RFC 4406. `http://www.faqs.org/rfcs/rfc4406.html`, 2006.

[63] U. Manber. Finding Similar Files in a Large File System. In *Proceedings of the Winter USENIX Technical Conference*, 1994.

[64] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[65] V. Metsis, I. Androutsopoulos, and G. Paliouras. Spam Filtering with Naive Bayes - Which Naive Bayes? In *Proceedings of the International Conference on Email and Anti-Spam (CEAS)*, 2006.

[66] J. Metzger, M. Schillo, and K. Fischer. A Multiagent-Based Peer-to-Peer Network in Java for Distributed Spam Filtering. In *Proceedings of the Third International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS)*, 2003.

[67] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-bandwidth Network File System. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, 2001.

[68] C. Neustaedter, A. J. B. Brush, and M. A. Smith. Beyond "From" and "Received": Exploring the Dynamics of Email Triage. In *Proceedings of the Conference On Human Factors In Computing Systems (CHI)*, 2005.

[69] C. Neustaedter, A. J. B. Brush, M. A. Smith, and D. Fisher. The Social Network and Relationship Finder: Social Sorting for Email Triage. In *Proceedings of the Second Conference on E-mail and Anti-Spam (CEAS)*, 2005.

[70] J. Oliver. Using Lexigraphical Distancing to Block Spam. In *Proceedings of the Spam Conference*, 2005.

[71] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based Runtime Software Evolution. In *Proceedings of 20th International Conference on Software Engineering (ICSE)*, 1998.

[72] P. Pantel and D. Lin. SpamCop: A Spam Classification & Organization Program. In *Proceedings of the AAAI Workshop on Learning for Text Categorization*, 1998.

[73] P. Pantel and D. Lin. SpamCop: A Spam Classification & Organization Program. In *Proceedings of the AAAI Workshop on Learning for Text Categorization*, 1998.

[74] V. V. Prakash and A. O'Donnell. Fighting Spam with Reputation Systems. *ACM Queue*, 3(9), 2005.

[75] M. B. Prince, L. Holloway, and A. M. Keller. Understanding How Spammers Steal Your E-Mail Address: An Analysis of the First Six Months of Data from Project Honey Pot. In *Proceedings of the Second Conference on Email and Anti-Spam (CEAS)*, 2005.
`http://www.projecthoneypot.org`.

[76] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, 2002.

[77] J. D. M. Rennie. ifile: An Application of Machine Learning to E-Mail Filtering. In *Proceedings of the Knowledge Discovery and Data Mining Workshop on Text Mining*, 2000.

[78] I. Rigoutsos and T. Huynh. Chung-Kwei: a Pattern-discovery-based System for the Automatic Identification of Unsolicited E-mail Messages (SPAM). In *Proceedings of the First Conference on Email and Anti-Spam (CEAS)*, 2004.

[79] G. Robinson. A Statistical Approach to the Spam Problem. *Linux Journal*, 2003.
`http://www.linuxjournal.com/article/6467`.

[80] S. L. Rohall, D. Gruen, P. Moody, M. Wattenberg, M. Stern, B. Kerr, B. Stachel, K. Dave, R. Armesa, and E. Wilcox. ReMail: a reinvented email prototype. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, 2004.

[81] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A Bayesian Approach to Filtering Junk E-Mail. In *Proceedings of the AAAI Workshop on Learning for Text Categorization*, 1998.

[82] G. Sakkis, I. Androutsopoulos, G. Paliouras, V. Karkaletsis, C. D. Spyropoulos, and P. Stamatopoulos. Stacking Classifiers for Anti-Spam Filtering of E-Mail. In *Proceedings of the 6th Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2001.

[83] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Analysis of Recommendation Algorithms for E-Commerce. In *Proceedings of ACM Conference on Electronic Commerce (EC)*, 2000.

[84] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2003.

[85] K. Schneider. Real-Time Spam URL Filtering. In *Proceedings of the Spam Conference*, 2004.

[86] A. Schwarz. *SpamAssassin - The Open Source Solution to SPAM.* O'Reilly, 2004.

[87] R. Segal. Classifier Aggregation. In *Proceedings of the Spam Conference*, 2005.

[88] R. Segal, J. Crawford, J. Kephart, and B. Leiba. SpamGuru: An Enterprise Anti-Spam Filtering System. In *Proceedings of the First Conference on E-mail and Anti-Spam (CEAS)*, 2004.

[89] N. Shivakumar and H. Garcia-Molina. SCAM: A Copy Detection Mechanism for Digital Documents. In *Proceedings of the Conference on the Theory and Practice of Digital Libraries*, 1995.

[90] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001.

[91] S. J. Stolfo, S. Hershkop, K. Wang, O. Nimeskern, and C.-W. Hu. Behavior Profiling of Email. In *Proceedings of the Symposium on Intelligence & Security Informatics (ISI)*, 2003.

[92] T. Sullivan. The More Things Change: Volatility and Stability in Spam Features. In *Proceedings of the Spam Conference*, 2004. `http://www.qaqd.com/research/mit04sum.html`.

[93] G. L. Wittel and S. F. Wu. On Attacking Statistical Spam Filters. In *Proceedings of the First Conference on Email and Anti-Spam (CEAS)*, 2004.

[94] B. Wu and B. D. Davison. Identifying Link Farm Spam Pages. In *Proceedings of the International World Wide Web Conference*, 2005.

[95] C.-C. Yeh and C.-H. Lin. Near-Duplicate Mail Detection Based on URL Information for Spam Detection. In *Proceedings of the International Conference on Information Networking (ICOIN)*, 2006.

[96] W. S. Yerazunis. The Spam-Filtering Accuracy Plateau at 99.9% Accuracy and How to Get Past It. In *Proceedings of the Spam Conference*, 2004.
`http://crm114.sourceforge.net/Plateau_Paper.pdf`.

[97] J. A. Zdziarski. Bayesian Noise Reduction: Progressive Noise Logic for Statistical Language Analysis. In *Proceedings of the Spam Conference*, 2005.

[98] J. A. Zdziarski. *Ending Spam*. No Starch Press, 2005.

[99] F. Zhou, L. Zhuang, B. Y. Zhao, L. Huang, A. D. Joseph, and J. Kubiatowicz. Approximate Object Location and Spam Filtering on Peer-to-Peer Systems. In *Proceedings of the ACM Middleware*, 2003.

Web References

[100] The Story of the Avalon Containers.
http://wiki.apache.org/avalon/ContainerStory.

[101] In the Fight Against Spam E-Mail, Goliath Wins Again.
http://www.washingtonpost.com/wp-dyn/content/article/2006/05/16/AR2006051601873.html.

[102] Bogofilter.
http://www.bgl.nu/bogofilter.

[103] M. Cegłowski and J. Schachter. Loaf.
http://loaf.cantbedone.org.

[104] D. Clinton. Schrödingers Collaborative Spam Filter.
http://blog.unto.net/work/schrdingers-collaborative-spam-filter.

[105] Cloudmark - Anti Spam and Spam Blocker Solutions.
http://www.cloudmark.com.

[106] The CRM114 Discriminator.
http://crm114.sourceforge.net.

[107] Docol©c - Plagiarism search.
http://www.docoloc.de.

[108] DSPAM.
http://dspam.nuclearelephant.com.

[109] Eclipse Foundation.
http://www.eclipse.org.

[110] J. Farmer. SpamPal.
http://www.spampal.org.

[111] Federal Energy Regulatory Commission. Information Released in Enron Investigation.
http://fercic.aspensys.com/members/manager.asp.

[112] Apache Excalibur Fortress.
http://excalibur.apache.org/fortress.

[113] GeoCities.
http://www.geocities.com.

[114] P. Graham. A Plan for Spam.
http://paulgraham.com/spam.html.

[115] J. Graham-Cumming. POPFile.
http://popfile.sourceforge.net.

[116] J. Graham-Cumming. The Spammers' Compendium.
http://www.jgc.org/tsc/.

[117] RFC: Internet Message Access Protocol – Version 4rev1.
http://www.ietf.org/rfc/rfc3501.txt.

[118] Institute of Electrical and Electronics Engineering. IEEE 754: Standard for Binary Floating-Point Arithmetic.
http://grouper.ieee.org/groups/754.

[119] B. Kamens. Bayesian Filtering: Beyond Binary Classification.
http://www.fogcreek.com/FogBugz/Downloads/KamensPaper.pdf.

[120] MIMEDefang.
http://www.mimedefang.org.

[121] MrPostman.
http://mrpostman.sourceforge.net.

[122] NanoContainer.
http://nanocontainer.codehaus.org.

[123] Nilsimsa.
http://ixazon.dynip.com/~cmeclax/nilsimsa.html.

[124] NiX Spam.
http://www.heise.de/ix/nixspam/x.

[125] Y. Oka. FQDN – fully qualified domain name list.
http://a-f.jp/oka/domains/fqdn2domain.

[126] The Okopipi Project.
http://www.okopipi.org.

[127] OSGi Alliance.
http://www.osgi.org.

[128] P. Perry. Resources on Collaborative Filtering.
http://www.paulperry.net/notes/cf.asp.

[129] PicoContainer.
http://picocontainer.codehaus.org.

[130] RFC: Post Office Protocol – Version 3.
http://www.ietf.org/rfc/rfc1939.txt.

[131] Postini. Hackers increase instant messaging attacks and use of image spam.
http://www.postini.com/news_events/pr/pr071406.php.

[132] Procmail.
http://www.procmail.org.

[133] Ling-Spam, PU, and Enron Email Corpora.
http://iit.demokritos.gr/skel/i-config/downloads.

[134] Pyzor.
http://pyzor.sourceforge.net.

[135] Vipul's Razor.
http://razor.sourceforge.net.

[136] [Razor-users] Revoke submits whole e-mail?
http://sourceforge.net/mailarchive/message.php?msg_id=9854948.

[137] Rhyolite Software. DCC - Distributed Checksum Clearinghouse.
http://www.rhyolite.com/anti-spam/dcc.

[138] US Secure Hash Algorithm 1 (SHA1).
http://www.faqs.org/rfcs/rfc3174.html.

[139] SORBS - Spam and Open-Relay Blocking System.
http://www.nl.sorbs.net.

[140] The Apache SpamAssassin Project.
http://spamassassin.apache.org.

[141] The Spamhaus Project.
http://www.spamhaus.org.

[142] W. Stearns. Razor2 protocol specification.
http://www.stearns.org/razor-caching-proxy/razor2-protocol.

[143] D. Streblechenko. Outlook Redemption.
http://www.dimastr.com/redemption.

[144] SURBL - Spam URI Realtime Blocklists.
http://www.surbl.org.

[145] J. Thornton. Collaborative Filtering Research Papers.
http://jamesthornton.com/cf.

[146] TinyURL.
http://www.tinyurl.com.

[147] Apache Tomcat.
http://tomcat.apache.org.

[148] Usenet Binary Spam Filter.
http://archive.xusenet.com/ubsf.html.

[149] URIBL - URI Blacklist.
http://www.uribl.com.

[150] Different ways to spell Viagra.
http://cockeyed.com/lessons/viagra/viagra.html.

[151] Wikipedia. Collaborative Filtering.
http://en.wikipedia.org/wiki/Collaborative_filtering.

[152] XULPlanet.
http://www.xulplanet.com.

[153] J. Zieren. Camel's Eye.
http://zieren.de/index.php?page=camelseye.

Curriculum Vitae

June 4, 1977	Born in Hamburg, Germany
1983–1996	Primary and high schools in Norderstedt and Monheim, Germany
1996–2002	Studies in computer science, University of Dortmund, Germany
January 2002	Diploma in computer science, University of Dortmund, Germany
2002–2006	Ph.D. student, research and teaching assistant, Distributed Computing Group, Prof. Roger Wattenhofer, ETH Zurich, Switzerland
September 2006	Ph.D. degree, Distributed Computing Group, ETH Zurich, Switzerland Advisor: Prof. Roger Wattenhofer Co-examiners: Prof. Gordon V. Cormack, University of Waterloo, Canada Prof. Christof Fetzer, University of Dresden, Germany