AntispamLab^{TM} - A Tool for Realistic Evaluation of Email Spam Filters

Slavisa Sarafijanovic [*]
EPFL, Switzerland
slavisa.sarafijanovic@epfl.ch

Luis Hernandez
UPC, Spain
lher8089@alu-etsetb.upc.edu

Raphael Naefen
EPFL, Switzerland
raphael.naefen@epfl.ch

Jean-Yves Le Boudec
EPFL, Switzerland
jean-yves.leboudec@epfl.ch

ABSTRACT

The existing tools for testing spam filters evaluate a filter instance by simply feeding it with a stream of emails, possibly also providing a feedback to the filter about the correctness of the detection. In such a scenario the evaluated filter is disconnected from the network of email servers, filters, and users, which makes the approach inappropriate for testing many of the filters that exploit some of the information about spam bulkiness, users' actions and social relations among the users. Corresponding evaluation results might be wrong, because the information that is normally used by the filter is missing, incomplete or inappropriate.

In this paper we present a tool to test spam filters in a very realistic scenario. Our tool consists of a set of Python scripts for Unix/Linux. The tool takes as inputs the filter to be tested and an affordable set of interconnected machines (e.g., PlanetLab machines, or locally created virtual machines). When started from a central place, the tool uses the provided machines to build a network of real email servers, installs instances of the filter, deploys and runs simulated email users and spammers, and computes the detection results statistic. Email servers are implemented using Postfix, a standard Linux email server. Only per-email-server filters are currently supported; testing per-email-client filters would require additional development of the tool. The size of the created emailing network is constrained only by the number of available PlanetLab or virtual machines. The run time is much shorter then the simulated system time, due to a time scaling mechanism. Testing a new filter is as simple as installing one copy of it in a real emailing network, which unifies the jobs of a new filter development, testing and prototyping. As an example of how to use the tool, we test the SpamAssassin filter.

1. INTRODUCTION

1.1 Need for Better Spam-Filters Evaluation Methods and Tools

Currently used methodology and tools for testing spam filters are based on so-called batch testing methods and online testing methods, borrowed from a mature and more general machine learning framework. A good survey of these machine-learning testing techniques can be found in [10].

Batch (a.k.a. off-line) testing is the most simplified method for evaluating spam filters. It is called "off-line" because it separates the learning and testing phases. Used corpora of labelled spam and good (a.k.a. ham) emails is, prior to the testing, divided into the two sets, a training set and a testing set. The filter is first trained using the first set, and then tested using the second set. Sometimes a third, so-called validation set is created from the corpora and used to optimize the filter's parameters during the training (in order to avoid over-training and preserve the generality of learning).

Existing incremental (a.k.a. online) testing methods feed a filter isolated from any emailing network with a stream of emails and, immediately or with a delay, provide the feedback to the filter about the correctness of the detection for each checked email.

Cormack and Bratko [4] show that cross-validation testing (a batch technique) and on-line testing results are different, suggesting that the validity of batch-tests results available in the literature is under question. In the same paper the authors investigate how the use of preprocessed-email databases, obtained by tokenization and/or obfuscation of emails [1, 2], affects testing results as compared to tests with a raw-email database. They show that pre-processing substantially changes the evaluation results for many filters. The overall conclusion that comes from their work is that using filter inputs and testing methodology - that are as realistic as possible - is important for proper testing and comparison of spam filters.

The TREC Spam Evaluation Toolkit [5] is probably the most realistic and widely used, publicly available, testing tool to date. It uses a classic online-testing method that models the filter input only by a simple stream of messages and the feedback about the correctness of their classification. We may say that this tool is the defacto standard for the evaluation and comparison of spam filters. Having a standard tool for testing spam filters is important, as it makes the comparison of a new filter with those evaluated and presented in the literature more easy, manageable and meaningful.

Despite its wide acceptance, the TREC tool does not account for the real emailing network scenario in which a filter might obtain and use certain inputs that come from the whole network of email servers, users and other filters. For example, if a filter exploits the bulkiness of DCC-like similarity-signatures derived from emails [6], the filtering results may depend on the ongoing spam pattern, i.e. on the

[*]Contact author.

CEAS 2007 - Fourth Conference on Email and Anti-Spam, August 2-3, 2007, Mountain View, California USA

content and timing of the emails observed by other users and filters; and they might also depend on the explicit or implicit feedback from different users about spam messages they receive. Such inputs are not properly provided by the TREC tool[1].

There are many other inputs, beside those listed in the example above, that are used by some existing filters or could be used by future filters and that are not well or not at all represented with the currently available testing tools.

1.2 Our Approach to Providing More Realistic Inputs to Tested Filters

It seems to us that the only way to feed an arbitrary filter with proper and realistic inputs is to provide a testing environment that properly represents all elements from a real emailing network: emails in their original syntax; users with their behavior and communication patterns; spamming techniques; communication between servers, filters, and email-reading programs; and timing of the events in the whole emailing network. Different approaches are possible in order to try to achieve this generally stated requirement.

The approach of having one isolated copy of the tested filter (as the TREC tool does) and of simulating any inputs the filter would normally use from the network has its drawbacks. Testing a filter that uses a new collaborative antispam technique would require a modelling and programming work in order to simulate the filter inputs that come from other filters and/or users. Moreover, modelling and simulation simplifications may be necessary in many cases and, if not done carefully, may lead to inaccurate filter inputs and inaccurate testing results. It seems that it could be much simpler and safer to try using multiple copies of the tested filter and directly generating such inputs. The complex phenomena that arise from the interaction of many elements are sometimes easier to recreate than to understand and model properly.

Another approach would be to simulate all the emailing network elements. Simulating a network of email servers itself and providing interfaces from each server to spam filters and email-reading programs seems to be feasible: The only functions of the network of email servers is to physically transfer emails from senders to the recipients with a variable (and usually small) delay and to do some simple additional processing jobs such as adding headers, handling non-existing recipients, deciding whether to relay emails, and forwarding emails to a filter if configured to do so. These functions alone seem to be simple to simulate within one machine. It also seems to be straightforward to add to such a simulator functions of email-reading programs, email users, and spamming.

On the contrary, when the inclusion of a filter to be tested is considered, it seems that the above mentioned "simulate everything" approach comes also with many difficulties and drawbacks. One drawback is related to the support of stateful filters, i.e. the filters that keep some state about protected accounts (most of the existing filters are stateful). This support would require each filter instance to be run separately, possibly leading to the CPU and active memory scalability problems. Disk space could also be a problem, because the memory space of the email accounts of all the users would have to be instantiated on the machine on which the simulation runs.

As it seems that building a one-machine simulator of the whole emailing network would have scalability problems, making this simulator distributed would probably be required in order to support emailing networks of meaningful size. Further, this would require implementing some intercommunication among the distributed simulator parts, under the constraint that the whole distributed simulator should still keep the simulated events properly ordered, which again increases implementation difficulties and the risk of having some undesired simplifications. Though simulating the functions of a network of email servers within one machine seems to be simple, implementing these functions within a distributed simulator might not be easy, may require some unwanted simplifications, and may cause some unexpected side effects.

Due to the above discussed possible drawbacks of the "simulate everything" approach, we decided to not simulate the network of email servers, but to simply build it by running multiple instances of real email servers. In order to keep the testing system as realistic as possible, but also in order to make it compatible with different environments in which it can be deployed and used, we decide to deploy (install) each email server on a separate virtual machine (VM). The complete testing system is deployed either on an already available set of VMs, such is PlanetLab [7], or VMs can be created locally on a provided set of real machines.

Installing each email server on a separate VM enables standard (and thus fully realistic) email server installation, configuration, and inter-communication; it also provides compatibility between our tool and the operating system of the machines on which the testing system is to be deployed and used.

We deploy other testing system components also on the VMs, for the same reasons as with the email servers. An email user and his email-reading program are simulated by a Python script instance that is put on a separate VM for each user. Spamming is simulated using the same approach as for simulating email users.

To implement the tool, we use shell scripting and Python [9], because the two together support very well networking tasks, working with files and automating the procedures. We automate all the testing procedures: deploying of the system, running tests, and collecting and processing the results.

1.3 Organization of The Paper

In the next section (Section 2) we explain the requirements and typical steps for using our testing tool, and we explain what the tool does, i.e. which services for evaluating spam filters it provides. In Section 3 we explain how we implement the tool and how the tool does its job. In Section 4 we illustrate use of the tool, by testing the SpamAssassin filter. In Section 5 we conclude the work presented in this paper. In Section 6 we discuss future work proposals.

[1]Actually the TREC tool has unused potential for supporting more realistic tests in certain cases: some inputs from the network to the filter could be simulated and added to the TREC tool, like answers to the DCC queries, for example. However, such an approach to keep one tested filter and to simulate any inputs it would normally use from the network has its drawbacks (discussed within Section 1.2).

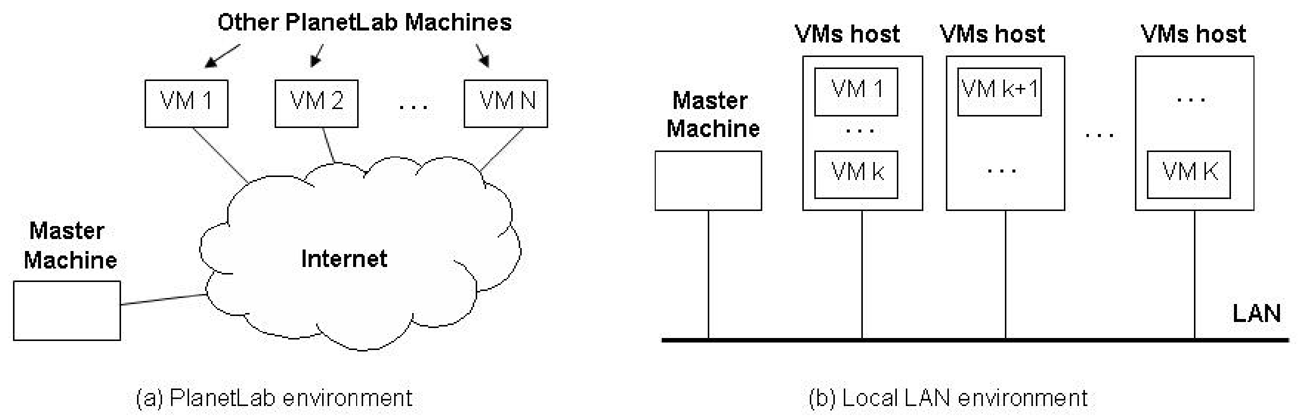

Figure 1: THE AntispamLab TOOL CAN BE USED IN TWO TESTING ENVIRONMENTS: (a) In the PlanetLab environment, the tool is run from one PlanetLab machine (master machine), and it sees each other PlanetLab machine i just as a virtual machine (VM i) reachable over the Internet. The tool deploys the testing system onto these VMs (i.e., it creates a network of email servers, filters, and simulated users and spammers), runs tests, and collects and processes the filtering results. To use the tool in this environment, one must have a PlanetLab user account.
(b) In the local-LAN environment, the user of the tool needs to create multiple VMs using the VM image provided with the tool, and to run a local DNS server that maps the names of the created VMs into their IP addresses. For this to work, the used real machines are required to support Xen, an open-source virtualization software for Linux/Unix. As the delay in the Internet is very small compared to the intervals between an email user events, both environments should provide very similar testing conditions.

2. WHAT THE TOOL DOES AND HOW IT CAN BE USED

The purpose of this section is to give, without going into the implementation details, more information to people interested in testing spam filters and wanting to decide whether our tool would serve their needs and what they would have to have and do in order to use the tool.

2.1 Requirements for Using The Tool

As shown and explained in Figure 1, the main requirement to use the tool is to either have a PlanetLab [7] account[2], or to provide a set of local-LAN machines with Xen [12] virtualization software support. PlanetLab is mainly available for university researchers. Local-LAN environment can be made using dedicated machines, or using spare CPU and memory of already existing machines. The only requirements for the local-LAN machines are to allow remote ssh access, and to allow running Xen and starting VMs over the obtained ssh terminal. The number of VMs that can be started on a real machine depends on the available RAM memory and disk space. With the current implementation, it is needed 256 MB of RAM and 4 GB of disk space per VM[3].

2.2 Services Provided by The Tool

Automated Deployment of The Testing System. Deploying the complete testing system, which includes the installation of multiple email servers, filters, users and spammers on the set of provided networked virtual machine, is completely automated by running only one tool script.

Automated Test Runs and Results Processing. Performing a test that consists of multiple independent runs of the testing system, collecting all the filtering results to a central place (from which the test is started), and processing them to get the confidence intervals for the true positive and false positive detection rates is also completely automated by running only one tool script.

Supported Filters. The tool supports two major standard interfaces used by per-email-server filters: SMTP and pipe. It does not currently support testing of per-email-client filters. Another constraint is that only the filters that can be installed on Linux/Unix are currently supported.

Supported Filter Inputs. The architecture of the testing system provides almost all filter inputs that exist in a real email-network: emails in their original syntax and with headers that correspond to the current communication patterns and timing in the simulated emailing network; feedbacks from the users; any other inputs that might come from the simulated network of email servers, filters, users, and spammers. The current implementation does not support all these inputs correctly. For example, the emails sent and received by a user are random samples (from the database) and do not represent well the profile of a real user (the profile might be used by some filters). Solving such problems requires some additional implementation work.

Time Scaling for Fast Testing. The tool implements a time scaling mechanism (explained in Section 3.4) that allows the run time of the tests to be much shorter then the simulated time, without affecting the testing results[4].

2.3 Using and Checking The Tool

To use the AntispamLab tool, one should first download it from [3] to one VM from the provided VMs set. Thereafter that VM is called the master machine.

[2]The user of a PlanetLab account is assigned a VM on each real PlanetLab machine that she/he wants to use. So the user owns by himself a very large network of VMs.

[3]These numbers can be considerably lowered by using a lighter image for running the VMs.

[4]Actually this is true for majority of the filters, but it is not for those that take into the account the time values present in the headers, in which case the tests should be run with the time scale equal to 1 (no scaling).

The tool contains two simple filter examples: one SMTP and one pipe filter. The tool is configured with the default test parameters, which define the filter to be tested (the pipe example filter), the emailing network size, the number runs within the test, simulated time of one run, time scale, etc. In order deploy the default testing system and to perform the default test (within the PlanetLab), the user of the tool has to provide the following inputs to the tool: his username on the PlanetLab, a list of PlanetLab nodes associated to his account and his PlanetLab key for remote ssh (PlanetLab documentation explains how to get this key). Such inputs also need to be provided if the tool is to be used on the VMs created locally (i.e., created using the VM image provided with the tool).

The two main scripts (for deploying the testing system and for performing a test) report to the screen of the master machine certain details on the deployment or test progress, which might be useful for locating problems if something goes wrong.

Once the tool is deployed and the default test is performed, the results of the test may be used for checking if the system works as expected. The checking is possible because the default example filter marks an email as spam with a given probability that depends on whether the email is really spam (which is known by the testing system). That means that the expected detection results are known for this filter. If the confidence intervals of the observed detection results contain the expected detection results, the testing system probably works well[5].

2.4 Testing A New Filter

As already mentioned, our tool supports only per-email-server filters that can be installed on a Linux/Unix. As installing such a filter requires (in principle) only the installation files to be provided and the installation commands to be run, we were able to automate the deployment and testing of a new filter.

A user wishing to test a new filter that uses one of the two supported interfaces should provide: all the files needed to install the filter (zipped under the name defined by our tool and copied into the defined tool folder); list of commands used to install and start the filter (written into a shell-script file of the tool); the type of the tested filter (SMTP or pipe) typed as a parameter into the main configuration file of the tool. Once the test is started, the deployment of the files, the installation and starting of the filters is done automatically by the tool.

3. HOW THE TOOL WORKS, AND HOW IT IS IMPLEMENTED

3.1 Deployment of A Testing System and A Test Execution

As PlanetLab nodes are seen by our deployment tool as VMs, in both PlanetLab and local-LAN scenarios the deployment procedure is the same. In Figure 2 it is explained how the tool deploys the testing system and how it controls a test execution. Sections 3.2-3.4 contain the additional details about how different testing system components and features are implemented.

[5]Though this checking does not prove the correctness of the testing system, it gives a strong indication that everything runs as expected. This checking helped use to discover few bugs in the system during its development.

3.2 Implementation of Main Components

Use of Real Email Servers. The tool installs and uses multiple instances of the Postfix [8] email server, a well known and widely used Linux email server. Each instance is installed and started via a small set of commands executed by a tool script (the tool also provides and installs certain packages needed to install and start Postfix, as these packages are not part of the PlanetLab VMs).

Simulating Email Users. The program that simulates an email user and its email-client program is written in Python. The functions of email-reading programs are simply implemented using the original IMAP function calls available in Python. The currently implemented model of the user is very simple. The intervals between the events at which a user checks for new emails are i.i.d. and generated uniformly between zero and a maximum value that is a testing system parameter. The times at which the user sends emails are generated in the same way. The user selects the recipient(s) of sent emails randomly among all the users present in the system (excluding self).

Simulating Spamming Process. Spamming is modelled and simulated in the same way as a normal email user is, with the differences that in the spamming case the sent emails are taken from the spam and not from the ham database, and the average inter-sending time is much smaller then for a simulated normal email-user.

Modelling and Using Email Databases. Within our tool we provide the databases of spam and ham messages, which are taken from the SpamAssassin [11] repositories (we use spam repository 20030228_spam_2.tar.bz2 and ham repository 20021010_easy_ham.tar.bz2). The databases are simply two folders (spam and ham) containing enumerated raw email files, and can be easily replaced by using other email repositories.

3.3 Filter Inputs

Feedback from Email Users. We implement only one type of feedback from email users to filters, to serve as an example. This is illustrated and explained in Figure 3. Other explicit or implicit feedbacks can be implemented in a similar way (events of standard deleting of emails by email users without scrolling through their content, for example).

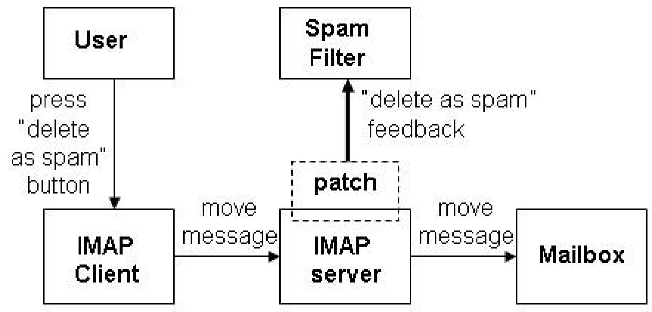

Figure 3: "DELETE AS SPAM" FEEDBACK FROM EMAIL USERS TO SPAM FILTERS: In order to detect triggered-by-user events of "deleting as spam" some messages and report them to the filter that protects the user, we patch IMAP server.

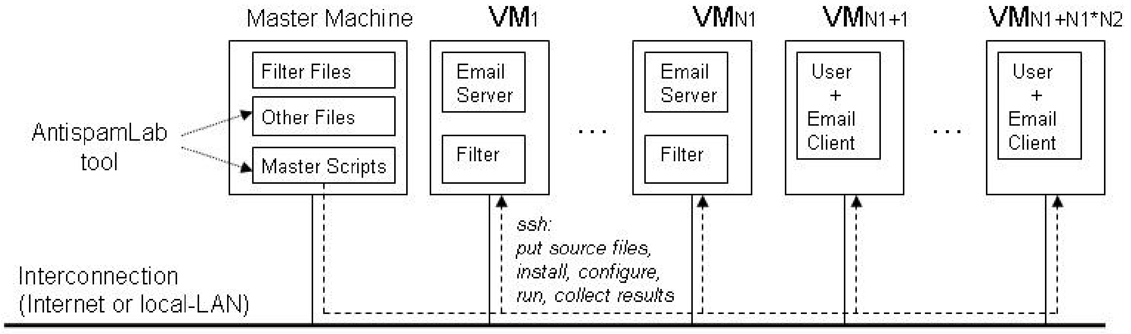

Figure 2: DEPLOYING THE TESTING SYSTEM AND PERFORMING A TEST. As explained in Figure 1, the testing system can be deployed on a provided set of virtual machines. Initially, all the files of the AntispamLab tool are put on one machine, which is then called the master machine (it can be a virtual machine (VM), or a real machine). There are two master scripts of the tool that can be executed from the master machine: the deploy script and the perform-a-test script. The deploy script first checks the availability of the VMs from the list of VMs provided by the user of the tool. Then it deploys the emailing network of N1 email servers and N2 email-users per server (N1 and N2 are configurable): It does ssh to N1+N1*N2 machines, puts on each machine the files needed for installing the corresponding components, runs email-servers and filters, and reports to the screen of the master machine whether the complete deployment was successful. In the case of a deployment failure on a VM, that VM is skipped and others are used (so the deploy script may try more then N1+N1*N2 machines). After a successful deployment, email servers and filters run, but email messages are not exchanged. Now the perform-a-test script can be executed. This script starts and controls a test, which consists of multiple runs of the complete system. It first sends the signals to the programs that simulate email users and clients to start. These programs are active for the duration of a test run (equal approximately to the simulated time divided by the time scale factor, see Section 3.4). Then they stop and send the detection reports (as observed by email users) to the perform-a-test script. After receiving all reports, the perform-a-test script makes the run log, cleans the data from the finished run, and starts another run. When all logs of all runs are collected (or some of them might time out and get discarded) the overall detection results with confidence intervals are computed and printed to the screen of the master machine.

Inputs from Other Filters. Our tool supports the deployment of multiple instances of a tested filter; these instances can communicate in the same way as they do in real life. The inputs that depend on other filters and users, but that are obtained from another networked antispam component (such is DCC) can be obtained by simply using an already existing such component, or by instantiating it (if its source code is available and the required resources are available) or a simulated copy of it within a VM by the user of the tool.

Support of Standard Email Format. The standard format of the emails fed to a tested filter, including correct header information that is adjusted to each separate run of the system, is naturally supported by the architecture of the testing system: The script that simulates a user and her/his emailing program may easily create appropriate header fields for each sent email, and use only the email content and the "Subject" field of the emails taken from spam and ham email repositories[6]; the email servers that relay the emails also add the appropriate header fields.

3.4 Time Scaling Mechanism

We exploit large empty slots that are typically present between the events in the testing system components, and we implement a simple time scaling mechanism that allows to better use available CPU of the VMs an speed up the testing. The basic idea is to scale-down the starting times of all the scheduled events using the same factor, but also to control the event collisions (that may be caused by this scaling) in a way that will prevent the scaling from affecting the testing results and still provide a considerable testing speedup. The way we implemented the time scaling mechanisms is illustrated and explained in Figure 4.

[6]Not yet completely supported by the current tool implementation.

4. AN EXAMPLE OF HOW TO TEST A NEW FILTER: TESTING SPAMASSASSIN

In order to test the SpamAssassin filter [11], we followed the general procedure explained in Section 2.4. We also follow the detailed procedure given in the README.txt file of the tool package, which points to the concrete files to be replaced.

We replace the default content of the file filter.tar.gz:

```
pipefilter.py
```

by the content that corresponds to SpamAssassin (we find the .rpm dependency-files needed for installing SpamAssassin by trying to install it once manually):

```
perl-Digest-HMAC-1.01-12.noarch.rpm
perl-Digest-SHA1-2.07-4.i386.rpm
perl-HTML-Parser-3.35-5.i386.rpm
perl-HTML-Tagset-3.03-29.noarch.rpm
perl-Net-DNS-0.45-3.i386.rpm
perl-Time-HiRes-1.55-2.i386.rpm
spamassassin-2.63-8.i386.rpm
```

We replace the default content of the file filter_install.sh:

```
mv pipefilter.py /usr/local
cp defaults.cfg /usr/local
```

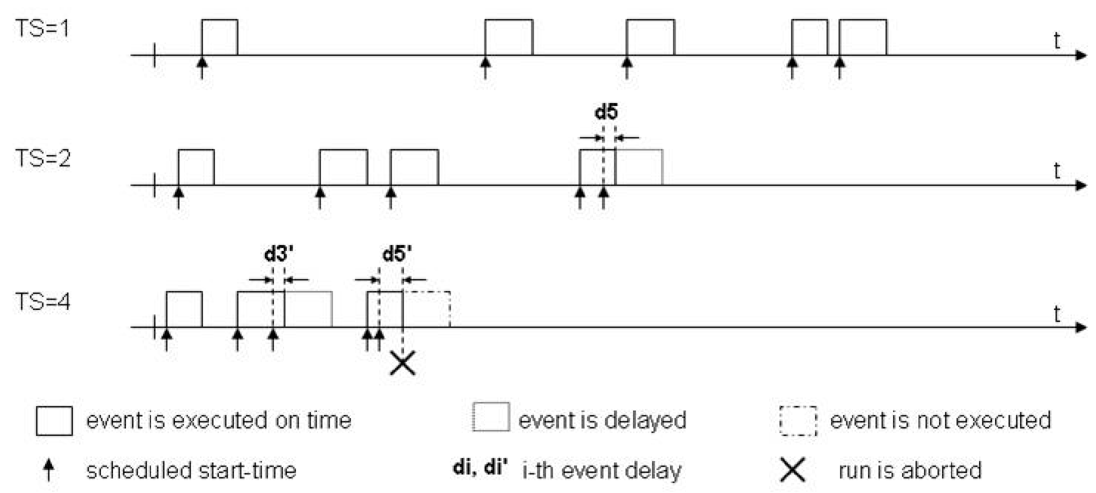

Figure 4: TIME SCALING MECHANISM. Start of each event (upper arrows) is scaled by the time scale factor TS. As this is done with all the independent events in the testing systems (and these are in our case only email-users), the relative order of the events stays approximately the same, and so the detection results should not change. The relative order is only approximately the same because the scaling of the events may lead to events overlaps and delays in cases when it would not happen without scaling. If these overlaps are small, they may be neglected (because there is also variable network delay which causes the same effect in both our testing system and in the real emailing networks). In the example the delay d3 is tolerated. The delay d5 is not tolerated because it is above a threshold, which will cause the run to be aborted (as previously explain a test consists of multiple runs; only successful runs are used for computing the detection statistic). In order to allow higher values of the time scaling factor and a better resistance of the tests to the abortions, we apply an additional rule that is not shown on this figure: we allow a predefined number of consecutive overlaps (a parameter) in order to give more chance to the system to recover in the cases when the delay is only temporary accumulating due to some short-time slow-down of one or more machines on which the system runs. The presence of possible side effects caused by the time scaling (that could impact the testing results) can always be checked using additional tests with TS=1. A typical value of TS that works well within the PlanetLab is 60.

by the content that corresponds to SpamAssassin (we find the commands needed for installing SpamAssassin in the original SpamAssassin instructions from SpamAssassin web page; as we use the SpamAssassin filter in the "pipe" mode, no command is needed to start the filter as a separate process):

```
rpm -i perl*
rpm -i spamassassin-2.63-8.i386.rpm
```

We set the simulated time to 3 hours, the time scale to 60, and the number of runs to 15. Other parameters of the test can be checked from the published tool source code, in the main configuration file setupvalues.cfg. We obtain the following test results:

```
Parsed 15 logs. Stat. relev: 15 (100.0%)
--------------------------------------
Mean TP ratio:  0.772794951866
Mean FP ratio:  0.0
--------------------------------------
[CI-lower , CI-upper] (seed=1)
TP:  [0.73441829584244445, 0.8043137254901962]
FP:  [0.0, 0.0]
```

The observed true positive detection is 0.77, with the confidence interval [0.73, 0.80]. False detections were not observed during the test. The execution time of each run was about 5 minutes (higher then 3 minutes due to the system restarts and feedback collecting). All the test details, along with the execution log, can be found on the AntispamLab web page [3].

5. CONCLUSION

In this paper, we have presented a new system for testing spam filters. We have implemented a tool that automates the deployment and use of the testing system. One important feature of the system is that it enables, thanks to its architecture, feeding of tested filters with very realistic inputs. For these possibilities to come fully into the effect, we need to improve some components of the tool, especially the models of email users and spamming, as the rest of the system represents well a real emailing network. Another important feature of the system is its compatibility to the operating system of the machines on which it can be used, which is achieved by use of the VMs and a standard (PlanetLab compatible) VM image.

The time-scaling mechanism provided within the tool is very useful for saving time when doing many evaluations of one or more filters. We also implement an example of the feedback from email users to spam filters (the "delete as spam" feedback), showing how implementing such feedbacks (from email users to spam filters) is convenient within our tool.

While the current implementation of the tool allows to automatically output the detection ratios observed for a tested filter, it is important to mention that realistic measuring of the CPU, memory and network traffic used by the tested filter is also well supported by the tool. To obtain these aditional performance metrics, one should simply make use of the operating system monitoring tools that are already available on the VMs on which the testing system gets deployed.

These additional metrics are needed, for example, in order to assess resistance of a spam filter or the whole emailing network to denial of service attacks by spammers. An example of such an attack is simply overloading the filters and the emailing network by randomly generated emails: this attack could make the email service temporary unavailable if, for example, it causes the filters to create the traffic for collaborative filtering that becomes too huge and congests the network. In a time when the availability is becoming a crucial requirement posed on the information systems, being able to observe these additional metrics is also becoming very important.

Automated tool use and its implementation with Python and shell scripts promise rapid further development of the tool and easy use of it. Another advantage of the tool is that the job of testing a new filter is approximately equivalent to the job of installing one copy of it in the real system, as we have illustrated on the SpamAssassin example.

In bottom line, we see the current design and implementation of the tool as a good framework toward more realistic testing of spam filters. As the tool recreates an emailing network, it might also be useful for testing other emailing network elements.

6. FUTURE WORK

The main future job is improving the models of email users and spamming. Email users should be modelled more completely by taking into account the facts such as: content-profiles of the users, the communication patterns regarding social contacts of the users, replying and forwarding of emails. While using the existing spam/ham databases, user modelling can still be considerably improved. Building and using more realistic and per-user specific spam/ham databases is a new requirement posed by our tool, which would enable modelling of the email users and their communication to be even more complete and realistic.

The modelling of the process of spamming should additionally take into account the facts such as: different content-obfuscation techniques used by spammers, sender-obfuscation techniques, use of botnets (the networks of hacked computers) for sending spam.

Another important future job is evaluating more spam filters and comparing the results of the tests obtained with our tool against the results of the tests obtained with other tools, in order to try to identify and show expected advantages or drawbacks of our testing system as compared to other testing systems.

The current implementation of our tool uses only Postfix as the email server. Supporting a windows-operating-system based email server, such as Exchange, would make the tool usable by a larger community of antispam people. We also plan to support the evaluation of per-email-client (i.e. per user) installed spam filters. These goal seem to be achievable by adding new modules to the existing implementation of the tool.

While the currently used and very simple time scaling mechanism already provides good testing speedup (60 times speedup worked very well for us), it doesn't prevent from loosing some runs of the test if a machine on which a component of the testing system runs becomes very slow. Instead of using simple time scaling and detection of critical event collisions, we consider use of the well known "virtual time" concept and an appropriate synchronization scheme in order to allow the simulated time in the testing system to advance as fast as it is allowed by the available computation resources.

In simulations done on a single machine, the"virtual time" concept means using a scheduler of all the events within the simulated system, and updating the simulated time and executing the next event immediately after the current event is finished. In our distributed system we could, for example, use a scheduler per testing system component, and re-synchronize the schedulers frequently enough in order to keep all the events within the testing system synchronized up to a small time error that is acceptable and known not to affect the tests significantly.

7. ACKNOWLEDGMENTS

The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation, under the grant number 5005-67322.

Special thanks to Marc Andre-Luthi, for his advice regarding some implementation choices and help in some wired "installation doesn't work" cases.

8. REFERENCES

[1] I. Androutsopoulos, J. Koutsias, K. Chandrinos, G. Paliouras, and C. Spyropoulos. An evaluation of naive bayesian anti-spam filtering. In Proc. of the workshop on Machine Learning in the New Information Age, 2000.

[2] I. Androutsopoulos, G. Paliouras, and E. Michelakis. Learning to filter unsolicited commercial e-mail. Tech. Rep. 2004/2, NCSR Demokritos, October 2004.

[3] AntispamLab. Project web page. http://lcawww.epfl.ch/ssarafij/antispamlab/, March 2007.

[4] G. V. Cormack and A. Bratko. Batch and online spam filter comparison. In *Conference on Email and Anti-Spam, CEAS 2006*, Mountain View, CA, July 2006.

[5] G. V. Cormack and T. R. Lynam. Overview of the trec 2005 spam evaluation track. In *Fourteenth Text REtrieval Conference (TREC-2005)*, Gaithersburg, MD, 2005.

[6] DCC. http://www.dcc-servers.net/dcc/, Feb 2007.

[7] PlanetLab. http://www.planet-lab.org/, Jan 2007.

[8] Postfix. http://www.postfix.org/, Jan 2007.

[9] Python. http://www.python.org/, Jan 2007.

[10] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, 2002.

[11] SpamAssassin. http://spamassassin.org/, Jan 2007.

[12] Xen. http://sourceforge.net/projects/xen/, Jan 2007.