B@bel: Leveraging Email Delivery for Spam Mitigation

Gianluca Stringhini[§], Manuel Egele[§], Apostolis Zarras[‡], Thorsten Holz[‡],
Christopher Kruegel[§], and Giovanni Vigna[§]

[§]University of California, Santa Barbara [‡] Ruhr-University Bochum
{gianluca,maeg,chris,vigna}@cs.ucsb.edu
{apostolis.zarras,thorsten.holz}@rub.de

Abstract

Traditional spam detection systems either rely on *content analysis* to detect spam emails, or attempt to detect spammers before they send a message, (i.e., they rely on the *origin* of the message). In this paper, we introduce a third approach: we present a system for filtering spam that takes into account *how* messages are sent by spammers. More precisely, we focus on the email delivery mechanism, and analyze the communication at the SMTP protocol level.

We introduce two complementary techniques as concrete instances of our new approach. First, we leverage the insight that different mail clients (and bots) implement the SMTP protocol in slightly different ways. We automatically learn these *SMTP dialects* and use them to detect bots during an SMTP transaction. Empirical results demonstrate that this technique is successful in identifying (and rejecting) bots that attempt to send emails. Second, we observe that spammers also take into account *server feedback* (for example to detect and remove non-existent recipients from email address lists). We can take advantage of this observation by returning fake information, thereby poisoning the server feedback on which the spammers rely. The results of our experiments show that by sending misleading information to a spammer, it is possible to prevent recipients from receiving subsequent spam emails from that same spammer.

1 Introduction

Email spam, or *unsolicited bulk email*, is one of the major open security problems of the Internet. Accounting for more than 77% of the overall world-wide email traffic [21], spam is annoying for users who receive emails they did not request, and it is damaging for users who fall for scams and other attacks. Also, spam wastes resources on SMTP servers, which have to process a significant amount of unwanted emails [41].

A lucrative business has emerged around email spam, and recent studies estimate that large affiliate campaigns generate between \$400K and \$1M revenue per month [20].

Nowadays, about 85% of world-wide spam traffic is sent by *botnets* [40]. Botnets are networks of compromised computers that act under the control of a single entity, known as the *botmaster*. During recent years, a wealth of research has been performed to mitigate both spam and botnets [18,22,29,31,33,34,50].

Existing spam detection systems fall into two main categories. The first category focuses on the *content* of an email. By identifying features of an email's content, one can classify it as spam or *ham* (i.e., a benign email message) [16,27,35]. The second category focuses on the *origin* of an email [17,43]. By analyzing distinctive features about the sender of an email (e.g., the IP address or autonomous system from which the email is sent, or the geographical distance between the sender and the recipient), one can assess whether an email is likely spam, without looking at the email content.

While existing approaches reduce spam, they also suffer from limitations. For instance, running content analysis on every received email is not always feasible for high-volume servers [41]. In addition, such content analysis systems can be evaded [25,28]. Similarly, origin-based techniques have coverage problems in practice. Previous work showed how IP blacklisting, a popular origin-based technique [3], misses a large fraction of the IP addresses that are actually sending spam [32,37].

In this paper, we propose a novel, third approach to fight spam. Instead of looking at the content of messages (what) or their origins (who), we analyze *the way* in which emails are sent (how). More precisely, we focus on the email delivery mechanism. That is, we look at the communication between the sender of an email and the receiving mail server at the SMTP protocol level. Our approach can be used in addition to traditional spam defense mechanisms. We introduce two complementary techniques as concrete instances of our new approach: *SMTP dialects* and *Server feedback manipulation*.

SMTP dialects. This technique leverages the observation that different email clients (and bots) implement the SMTP protocol in slightly different ways. These deviations occur at various levels, and range from differences in the case of protocol keywords, to differences in the syntax of individual messages, to the way in which messages are parsed. We refer to deviations from the strict SMTP specification (as defined in the corresponding RFCs) as *SMTP dialects*. As with human language dialects, the listener (the server) typically understands what the speaker (a legitimate email client or a bot) is saying. This is because SMTP servers, similar to many other Internet services, follow *Postel's law*, which states: "Be liberal in what you accept, and conservative in what you send."

We introduce a model that represents SMTP dialects as state machines, and we present an algorithm that learns dialects for different email clients (and their respective email engines). Our algorithm uses both passive observation and active probing to efficiently generate models that can distinguish between different email engines. Unlike previous work on service and protocol fingerprinting, our models are stateful. This is important, because it is almost never enough to inspect a single message to be able to identify a specific dialect.

Leveraging our models, we implement a decision procedure that can, based on the observation of an SMTP transaction, determine the sender's dialect. This is useful, as it allows an email server to terminate the connection with a client when this client is recognized as a spambot. The connection can be dropped before any content is transmitted, which saves computational resources at the server. Moreover, the identification of a sender's dialect allows analysts to group bots of the same family, or track the evolution of spam engines within a single malware family.

Server feedback manipulation. The SMTP protocol is used by a client to send a message to the server. During this transaction, the client receives from the server information related to the delivery process. One important piece of information is whether the intended recipient exists or not. The performance of a spam campaign can improve significantly when a botmaster takes into account server feedback. In particular, it is beneficial for spammers to remove non-existent recipient addresses from their email lists. This prevents a spammer from sending useless messages during subsequent campaigns. Indeed, previous research has shown that certain bots report the error codes received from email servers back to their command and control nodes [22, 38].

To exploit the way in which botnets currently leverage server feedback, it is possible to manipulate the responses from the mail server to a bot. In particular, when a mail server identifies the sender as a bot, instead of dropping the connection, the server could simply reply that the recipient address does not exist. To identify a bot, one can either use traditional origin-based approaches or leverage the SMTP dialects proposed in this paper. When the server feedback is poisoned in this fashion, spammers have to decide between two options. One possibility is to continue to consider server feedback and, as a result, remove valid email addresses from their email list. This reduces the spam emails that these users will receive in the future. Alternatively, spammers can decide to distrust and discard any server feedback. This reduces the effectiveness of future campaigns since emails will be sent to non-existent users.

Our experimental results demonstrate that our techniques are successful in identifying (and rejecting) bots that attempt to send unwanted emails. Moreover, we show that we can successfully poison spam campaigns and prevent recipients from receiving subsequent emails from certain spammers. However, we recognize that spam is an adversarial activity and an arms race. Thus, a successful deployment of our approach might prompt spammers to adapt. We discuss possible paths for spammers to evolve, and we argue that such evolution comes at a cost in terms of performance and flexibility.

To summarize, the paper makes the following main contributions:

- We introduce a novel approach to detect and mitigate spam emails. This approach focuses on the email delivery mechanism — the SMTP communication between the email client and the email server. It is complementary to traditional techniques that operate either on the message origin or on the message content.
- We introduce the concept of SMTP dialects as one concrete instance of our approach. Dialects capture small variations in the ways in which clients implement the SMTP protocol. This allows us to distinguish between legitimate email clients and spambots. We designed and implemented a technique to automatically learn the SMTP dialects of both legitimate email clients and spambots.
- We implemented our approach in a tool, called `B@bel`. Our experimental results demonstrate that `B@bel` is able to correctly identify spambots in a real-world scenario.
- We study how the feedback provided by email servers to bots is used by their botmasters. As a second instance of our approach, we show how providing incorrect feedback to bots can have a negative impact on the spamming effectiveness of a botnet.

2 Background: The SMTP Protocol

The Simple Mail Transfer Protocol (*SMTP*), as defined in RFC 821 [1], is a text-based protocol that is used to send email messages originating from *Mail User Agents* (MUAs — e.g., Outlook, Thunderbird, or Mutt), through intermediate *Mail Transfer Agents* (MTAs — e.g., Sendmail, Postfix, or Exchange) to the recipients' mailboxes. The protocol is defined as an alternating dialogue where the sender and the receiver take turns transmitting their messages. Messages sent by the sender are called *commands*, and they instruct the receiver to perform an action on behalf of the sender. The SMTP RFC defines 14 commands. Each command consists of four case-insensitive, alphabetic-character command codes (e.g., `MAIL`) and additional, optional arguments (e.g., `FROM:<me@example.com>`). One or more space characters separate command codes and argument fields. All commands are terminated by a line terminator, which we denote as <CR><LF>. An exception is the `DATA` command, which instructs the receiver to accept the subsequent lines as the email's content, until the sender transmits a dot character as the only character on a line (i.e., <CR><LF>.<CR><LF>).

SMTP *replies* are sent by the receiver to inform the sender about the progress of the email transfer process. Replies consist of a three-digit status code, followed by a space separator, followed by a short textual description. For example, the reply `250 Ok` indicates to the sender that the last command was executed successfully. Commonly, replies are one line long and terminated by <CR><LF>[1]. The RFC defines 21 different reply codes. These codes inform the sender about the specific state that the receiver has advanced to in its protocol state machine and, thus, allows the sender to synchronize its state with the state of the receiver. A plethora of additional RFCs have been introduced to extend and modify the original SMTP protocol. For example, RFC 1869 introduced *SMTP Service Extensions*. These extensions define how an SMTP receiver can inform a client about the extensions it supports. More precisely, if a client wants to indicate that it supports SMTP Service Extensions, it will greet the server with `EHLO` instead of the regular `HELO` command. The server then replies with a list of available service extensions as a multi-line reply. For example, a server capable of handling encryption can announce this capability by responding with a `250-STARTTLS` reply to the client's `EHLO` command.

MTAs, mail clients, and spambots implement different sets of these extensions. As we will discuss in detail later, we leverage these differences to determine the SMTP dialect spoken in a specific SMTP conversation.

```
Server: 220 debian
Client: HELO example.com
Server: 250 OK
Client: MAIL FROM:<me@example.com>
Server: 250 2.1.0 OK
Client: RCPT TO:<you@example.com>
Server: 250 2.1.5 OK
Client: DATA
```

Figure 1: A typical SMTP conversation

In this paper, we consider an *SMTP conversation* the sequence of commands and replies that leads to a `DATA` command, to a `QUIT` command, or to an abrupt termination of the connection. This means that we do not consider any reply or command that is sent after the client starts transmitting the actual content of an email. An example of an SMTP conversation is listed in Figure 1.

3 SMTP Dialects

The RFCs that define SMTP specify the protocol that a client has to speak to properly communicate with a server. However, different clients might implement the SMTP protocol in slightly different ways, for three main reasons:

1. The SMTP RFCs do not always provide a single possible format when specifying the commands a client must send. For example, command identifiers are case insensitive, which means that `EHLO` and `ehlo` are both valid command codes.
2. By using different SMTP extensions, clients might add different parameters to the commands they send.
3. Servers typically accept commands that do not comply with the strict SMTP definitions. Therefore, a client might implement the protocol in slightly wrong ways while still succeeding in sending email messages.

We call different implementations of the SMTP protocol *SMTP dialects*. A dialect **D** is defined as a state machine

$$\mathbf{D} = < \Sigma, S, s_0, T, F_g, F_b >,$$

where Σ is the input alphabet (composed of server replies), S is a set of states, s_0 is the initial state, and T is a set of transitions. Each state s in S is labeled with a client command, and each transition t in T is labeled with a server reply. $F_g \subseteq S$ is a set of good states, which represent successful SMTP conversations, while $F_b \subseteq S$ is a set of bad states, which represent failed SMTP conversations.

[1]The protocol allows the server to answer with multi-line replies. In a multi-line reply, all lines but the last must begin with the status code followed by a dash character. The last line of a multi-line reply must be formatted like a regular one-line reply

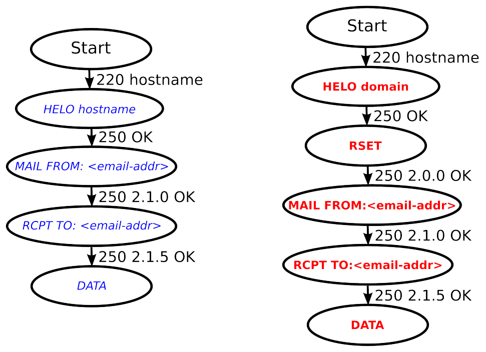

Figure 2: Simplified state machines for Outlook Express (left) and Bagle (right).

The state machine **D** captures the order in which commands are sent in relation to server replies by that particular dialect.

Since SMTP messages are not always constant, but contain variable fields (e.g., the recipient email address in an `RCPT` command), we abstract commands and replies as templates, and label states and transitions with such templates.

We do not require **D** to be deterministic. The reason for this is that some clients show a non-deterministic behavior in the messages they exchange with SMTP servers. For example, bots belonging to the *Lethic* malware family use `EHLO` and `HELO` interchangeably when responding to a server `220` reply. Figure 2 shows two example dialect state machines (Outlook Express and Bagle, a spambot).

3.1 Message Templates

As explained previously, we label states and transitions with message templates. We define the templates of the messages that belong to a dialect as regular expressions. Each message is composed of a sequence of tokens. We define a token as any sequence of characters separated by delimiters. We define spaces, colons, and equality symbols as delimiters. We leverage domain knowledge to develop a number of regular expressions for the variable elements in an SMTP conversation. In particular, we define regular expressions for email addresses, fully qualified domain names, domain names, IP addresses, numbers, and hostnames (see Figure 3 for details). Every token that does not match any of these regular expressions is treated as a keyword.

An example of a message template is

```
MAIL From:  <email-addr>,
```

where `email-addr` is a regular expression that matches email addresses.

Given two dialects **D** and **D'**, we consider them different if their state machines are different. For example, the two dialects in Figure 2 differ in the sequence of commands that the two clients send: Bagle sends a `RSET` command after the `HELO`, while Outlook Express sends a `MAIL` command directly. Also, the format of the commands of the two dialects differs: Outlook Express puts a space between `MAIL FROM:` and the sender email address, while Bagle does not.

```
Email address: <?[\w\.-]+@[\w\.-]+>?
IP address: \[?\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\]?
Fully qualified domain name: [\w-]+\.[\w-]+\.\w[\w-]+
Domain name: [\w-]+\.[\w-]+
Number: [0-9]{3}[0-9]+
Hostname: [\w-]{5}[\w-]+
```

Figure 3: Regular expressions used in message templates.

In Section 4, we show how we can learn the dialect spoken by an SMTP client. In Section 5, we show how these learned dialects can be matched against an SMTP conversation, which is crucial for performing spam mitigation, as we will show in Section 7.

4 Learning Dialects

To distinguish between different SMTP speakers, we require a mechanism that learns which dialect is spoken by a particular client. To do this, we need a set of SMTP conversations **C** generated by the client. Each conversation is a sequence of <*reply*, *command*> pairs, where *command* can be empty if the client did not send anything after receiving a reply from the server.

It is important to note that the state machine learned for the dialect is affected by the type of conversations in **C**. For example, if **C** only contains successful SMTP conversations, the portion of the dialect state machine that we can learn from it is very small. In the typical SMTP conversation listed in Figure 1, the client first connects to the SMTP server, then announces itself (i.e., sends a `HELO` command), states who the sender of the email is (i.e., sends a `MAIL` command), lists recipients (by using one or more `RCPT` commands), and starts sending the actual email content (by sending a `DATA` command). Observing this type of communication gives no information on what a client would do upon receiving a particular error, or a specific SMTP reply from the server. To mitigate this problem, we collect a diverse set of SMTP conversations. We do this by directing the client to an SMTP server under our control, and sending specific SMTP replies to it (see Section 4.2).

Even though sending specific replies allows us to explore more states than the ones we could explore otherwise, we still cannot be sure that the dialects we learn are complete. In Section 7, we show how the inferred state machines are usually good enough for discriminating between different SMTP dialects. However, in some cases, we might not be able to distinguish two different dialects because the learned state machines are identical.

4.1 Learning Algorithm

Analyzing the set **C** allows us to learn part of the dialect spoken by the client. Our learning algorithm processes one SMTP conversation from **C** at a time, and iteratively builds the dialect state machine.

4.1.1 Learning the Message Templates

For each message observed in a conversation Con in **C**, our algorithm generates a regular expression that matches it. The regular expression generation algorithm works in three steps:

Step 1: First, we split the message into tokens. As mentioned in Section 3.1, we consider the space, colon, and equality characters as delimiters.

Step 2: For each token, we check if it matches a known regular expression. More precisely, we check it against all the regular expressions defined in Figure 3, from the most specific to the least specific, until one matches (this means that we check the regular expressions in the following order: email address, IP address, fully qualified domain name, domain name, number, hostname).

If a token matches a regular expression, we substitute the token with the matched regular expression's identifier (e.g., `<email-addr>`). If none of the regular expressions are matched, we consider the token a keyword, and we include it verbatim in the template.

Step 3: We build the message template, by concatenating the template tokens (which can be keywords or regular expressions) and the delimiters, in the order in which we encountered them in the original message.

Consider, for example, the command:

```
MAIL FROM:<evil@example.com>
```

First, we break the command into tokens:

[`MAIL`, `FROM`, `<evil@example.com>`]

The only token that matches one of the known regular expressions is the email address. Therefore, we consider the other tokens as keywords. The final template for this command will therefore be:

```
MAIL FROM:<email-addr>
```

Notice that, by defining message format templates as we described, we can be more precise than the SMTP standard specification and detect the (often subtle) differences between two dialects even though both might comply with the SMTP RFC. For example, we would build two different message format templates (and, therefore, have two dialects) for two clients that use different case for the `EHLO` keyword (e.g., one uses `EHLO` as a keyword, while the other uses `Ehlo`).

4.1.2 Learning the State Machine

We incrementally build the dialect state machine by starting from an empty initial state s_0 and adding new transitions and states as we observe more SMTP conversations from **C**. For each conversation *Con* in **C**, the algorithm executes the following steps:

Step 1: We set the current state s to s_0.

Step 2: We examine all tuples $<r_i, c_i>$ in *Con*. An example of a tuple is <`220 server, HELO evil.com`>.

Step 3: We apply the algorithm described in 4.1.1 to r_i and c_i, and build the corresponding templates t_r and t_c. In the example, t_r is `220 hostname` and t_c is `HELO domain`. Note that c_i could be empty, because the client might not have sent any command after a reply from the server. In this case t_c will be an empty string.

Step 4: If the state machine has a state s_j labeled with t_c, we check if there is a transition t labeled with t_r going from s to s_j. (i) If there is one, we set the current state s to s_j, and go to Step 6. (ii) If there is no such transition, we connect s and s_j with a transition labeled with t_r, set the current state s to s_j, and go to Step 6. (iii) If none of the previous conditions hold, we go to Step 5.

Step 5: If there is no state labeled with t_c, we create a new state s_n, label it with t_c , and connect s and s_n with a transition labeled t_r. We then set the current state s to s_n. Following the previous example, if we have no state labeled with `HELO domain`, we create a new state with that label, and connect it to the current state s (in this case the initial state) with a transition labeled with `220 hostname`. If there are no tuples left in *Con*, and t_c is empty, we set the current state as a failure state for the current dialect, and add it to F_b. We then move to the next conversation in **C**, and go back to Step 2 [2]. Otherwise, we go to Step 6.

Step 6: If s is labeled with `DATA`, we mark the state as a good final state for this dialect, and add it to F_g. Else, if s is labeled with `QUIT`, we mark s as a bad final state and add it to F_b. We then move to the next conversation in **C**, and we go back to Step 2.

[2]By doing this, we handle cases in which the client abruptly terminates the connection

4.2 Collecting SMTP Conversations

To be able to model as much of a dialect as possible, we need a comprehensive set of SMTP conversations generated by a client.

As previously discussed, the straightforward approach to collect SMTP conversations is to passively observe the messages exchanged between a client and a server. In practice, this is often enough to uniquely determine the dialect spoken by a client (see Section 7 for experimental results). However, there are cases in which passive observation is not enough to uniquely identify a dialect. In such cases, it would be beneficial to be able to send specifically-crafted replies to a client (e.g., malformed replies), and observe its responses.

To perform this exploration, we set up a testing environment in which we direct clients to a mail server we control, and we instrument the server to be able to craft specific responses to the commands the client sends.

The SMTP RFCs define how a client should respond to unexpected SMTP replies, such as errors and malformed messages. However, both legitimate clients and spam engines either exhibit small differences in the implementation of these guidelines, or they do not implement them correctly. The reason for this is that implementing a subset of the SMTP guidelines is enough to be able to perform a correct conversation with a server and successfully send an email, in most cases. Therefore, there is no need for a client to implement the full SMTP protocol. Of course, for legitimate clients, we expect the SMTP implementation to be mature, robust, and complete — that is, corner cases are handled correctly. In contrast, spambots have a very focused purpose when using SMTP: send emails as fast as possible. For spammers, taking into account every possible corner case of the SMTP protocol is unnecessary; even more problematic, it could impact the performance of the spam engine (see Section 7.4 for more details).

In summary, we want to achieve two goals when actively learning an SMTP dialect. First, we want to learn how a client reacts to replies that belong to the language defined in the SMTP RFCs, but are not exposed during passive observation. Second, we want to learn how a client reacts to messages that are invalid according to the SMTP RFCs.

We aim to systematically explore the message structure as well as the state machine of the dialect spoken by a client. To this end, the variations to the SMTP protocol we use for active probing are of two types: (i) variations to the protocol state machine, which modify the sequence or the number of the replies that are sent by the server; and (ii) variations to the replies, which modify the structure of the reply messages that are sent by the server.

In the following, we discuss how we generate variations of both the protocol state machine and the replies.

Protocol state machine variations. We use four types of protocol variation techniques:
Standard SMTP replies: These variations aim at exposing responses to replies that comply with the RFCs, but are not observable during a standard, successful SMTP conversation, like the one in Figure 1. An example is sending SMTP errors to the commands a client sends. Some dialects continue the conversation with the server even after receiving a critical error.
Additional SMTP replies: These variations add replies to the SMTP conversation. More precisely, this technique replies with more than one message to the commands the client sends. Some dialects ignore the additional replies, while others will only consider one of the replies.
Out-of-order SMTP replies: These variations are used to analyze how a client reacts when it receives a reply that should not be sent at that point in the protocol (i.e., a state transition that is not defined by the standard SMTP state machine). For example, some senders might start sending the email content as soon as they receive a 354 reply, even if they did not specify the sender and recipients of the email yet.
Missing replies: These variations aim at exposing the behavior of a dialect when the server never sends a reply to a command.

Message format variations. These variations represent changes in the format of the replies that the server sends back to a client. As described in Section 2, SMTP server replies to a client's command have the format `CODE TEXT<CR><LF>`, where `CODE` represents the actual response to the client's command, `TEXT` provides human-readable information to the user, and `<CR><LF>` is the line terminator. According to the SMTP specification, a client should read the data from the server until it receives a line terminator, parse the code to check the response, and pass the text of the reply to the user if necessary (e.g., in case an error occurred).

Given the specification, we craft reply variations in four distinct ways to systematically study how a client reacts to them:
Compliant replies: These reply variations comply with the SMTP standard, but are seldom observed in a common conversation. For example, this technique might vary the capitalization of the reply (uppercase/lowercase/mixed case). The SMTP specification states that reply text should be case-insensitive.
Incorrect replies: The SMTP specification states that reply codes should always start with one of the digits 2, 3, 4, or 5 (according to the class of the status code), and be three-digits long. These variations are replies that do not comply with the protocol (e.g., a message with a re-

ply code that is four digits long). A client is expected to respond with a `QUIT` command to these malformed replies, but certain dialects behave differently.

Truncated replies: As discussed previously, the SMTP specification dictates how a client is supposed to handle the replies it receives from the server. Of course, it is not guaranteed that clients will follow the specification and process the entire reply. The reason is that the only important information the client needs to analyze to determine the server's response is the status code. Some dialects might only check for the status code, discarding the rest of the message. For these reasons, we generate variations as follows: For each reply, we first separate it into tokens as described in Section 3.1. Then, for each token, we generate N different variations, where N is the number of tokens in each reply. We obtain such variations by truncating the reply with a line terminator after each token.

Incorrectly-terminated replies: From a practical point of view, there is no need for a client to parse the full reply until it reaches the line terminator. To assess whether a dialect checks for the line terminator when receiving a reply, we terminate the replies with incorrect terminators. In particular, we use the sequences `<CR>`, `<LF>`, `<CR><CR>`, and `<LF><LF>` as line terminators. For each terminator, similar to what we did for truncated replies, we generate $4N$ different variations of each reply, by truncating the reply after every token.

We developed 228 variations to use for our active probing. More precisely, we extracted the set of replies that are contained in the Postfix [3] source code. Then, we applied to them the variations described in this section, and we injected them into a reference SMTP conversation. To this end, we used the sequence of server replies from the conversation in Figure 1.

5 Matching Conversations to Dialects

After having learned the SMTP dialects for different clients, we obtain a different state machine for each client. Given a conversation between a client and a server, we want to assess which dialect the client is speaking. To do this, we merge all inferred dialect state machines together into a single *Decision State Machine* $\mathbf{M}_D$.

5.1 Building the Decision State Machine

We use the approach proposed by Wolf [46] to merge the dialect state machines into a single state machine. Given two dialects $\mathbf{D}_1$ and $\mathbf{D}_2$, the approach works as follows:

Step 1: We build the Cartesian product $\mathbf{D}_1 \times \mathbf{D}_2$. That is, for each combination of states $< s_1, s_2 >$, where s_1 is a state in $\mathbf{D}_1$ and s_2 is a state in $\mathbf{D}_2$, we build a new state s_D in the decision state machine $\mathbf{M}_D$.

[3] A popular open-source Mail Transfer Agent: `http://www.postfix.org/`

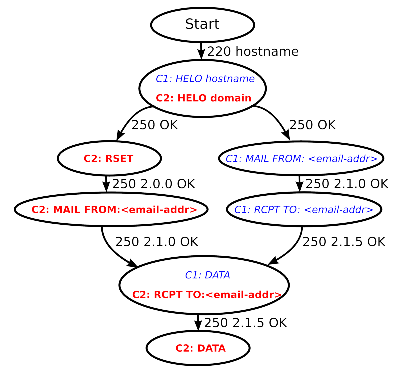

Figure 4: An example of decision state machine

The label of s_D is a table with two columns. The first column contains the identifier of one of the dialects s_D was built from (e.g., $\mathbf{D}_1$), and the second column contains the label that dialect had in the original state (either s_1 or s_2). Note that we add one row for each of the two states that s_D was built from. For example, the second state of the state machine in Figure 4 is labeled with a table containing the two possible message templates that the clients `C1` and `C2` would send in that state (i.e., `HELO hostname` and `HELO domain`).

We then check all the incoming transitions to s_1 and s_2 in the original state machines $\mathbf{D}_1$ and $\mathbf{D}_2$. For each combination of transitions $<t_1, t_2>$, where t_1 is an incoming transition for s_1 and t_2 is an incoming transition for s_2, we check if t_1 and t_2 have the same label. If they do, we generate a new transition t_d, and add it to $\mathbf{M}_D$. The label of t_d is the label of t_1 and t_2. The start state of t_d is the Cartesian product of the start states of t_1 and t_2, respectively, while the end state is s_D. If the labels of s_1 and s_2 do not match, we discard t_d. For example, a transition t_1 labeled as `250 OK` and a transition t_2 labeled as `553 Relaying Denied` would not generate a transition in $\mathbf{M}_D$. At the end of this process, if s_D is not connected to any other state, it will be not part of the decision state machines $\mathbf{M}_D$, since that state would not be reachable if added to $\mathbf{M}_D$.

Step 2: We reduce the number of states in $\mathbf{M}_D$ by merging together states that are *equivalent*. To evaluate if two states s_1 and s_2 are equivalent, we first extract the set of incoming transitions to s_1 and s_2. We name these sets I_1 and I_2. Then, we extract the set of outgoing transitions from s_1 and s_2, and name these sets O_1 and O_2. We consider s_1 and s_2 as equivalent if $|I_1| = |I_2|$ and $|O_1| = |O_2|$, and if the edges in the sets I_1 and I_2, and in O_1 and O_2 have the exact same labels.

If s_1 and s_2 are equivalent, we remove them from $\mathbf{M}_D$, and we add a state s_d to $\mathbf{M}_D$. The label for s_d is a table composed of the combined rows of the label tables of s_1 and s_2. We then adjust all the transitions in $\mathbf{M}_D$ that had s_1 or s_2 as start states to start from s_d, and all the transitions that had s_1 or s_2 as end states to end at s_d.

We iteratively run this algorithm on all the dialects we learned, and we build the final decision state machine $\mathbf{M}_D$. As an example, Figure 4 shows the decision state machine built from the two dialects in Figure 2. Wolf shows how this algorithm produces nearly-minimal resulting state machines [46]. Empirical results indicate that this works well in practice and is enough for our purposes. Also, as for the dialect state machines, the decision state machine is non-deterministic. This is not a problem, since we analyze different states in parallel to make a decision as we explain in the next section.

5.2 Making a Decision

Given an SMTP conversation Con, we assign it to an SMTP dialect by traversing the decision state machine $\mathbf{M}_D$ in the following way:

Step 1: We keep a list A of active states, and a list C_D of dialect candidates. At the beginning of the algorithm, A only contains the initial state of $\mathbf{M}_D$, while C_D contains all the learned dialects.

Step 2: Every time we see a server reply r in Con, we check each state s_a in A for outgoing transitions labeled with r. If such transition exists, we follow each of them and add the end states to a list A'. Then, we set A' as the new active state list A.

Step 3: Every time we see a client command c in Con, we check each state s_a in A. If s_a's table has an entry that matches c, and the identifier for that entry is in the dialect candidate list C_D, we copy s_a to a list A'. We then remove from C_D all dialect candidates whose table entry in s_a did not match c. We set A' as the new active state list A.

The dialects that are still in C_D at the end of the process are the possible candidates the conversation belongs to. If C_D contains a single candidate, we can make a decision and assign the conversation to a unique dialect.

5.3 Applying the Decision

The decision approach explained in the previous section can be used in different ways, and for different purposes. In particular, we can use it to assess to which client a server is talking. Furthermore, we can use it for spam mitigation, and close connections whenever a conversation matches a dialect spoken by a bot.

Similarly to what we discussed in Section 4, the decision process can happen passively, or actively, by having a server decide which replies to send to the client. In the first case, we traverse the decision state machine for each reply, as described in Section 5.2, and end up with a dialect candidate set at the end of the conversation. Consider, for example, the decision state machine in Figure 4. By passively observing the SMTP conversation, our approach is able to discard one of the two dialects from the candidate set as soon as the client sends the `HELO` message. If the commands of the remaining candidate match the ones in the decision state machine for that client until we observe the `DATA` command, we can attribute the conversation to that dialect. Otherwise, the conversation does not belong to any learned dialect.

As discussed in Section 4, passive observation gives no guarantee to uniquely identify a dialect. In this context, a less problematic use case is to deploy this approach for spam detection: once the candidate set C_D contains only bots, we can close the connection and classify this conversation as related to spam. As we will show in Section 7, this approach works well in practice on a real-world data set. If passive observation is not enough to identify a dialect, one can use active probing.

Gain heuristic. To perform active detection, we need to identify "good" replies that we can send to achieve our purpose (dialect classification or spam mitigation). More specifically, we need to find out which replies can be used to expose the deviations in different implementations. To achieve this goal, we use the following heuristic: For each state c_i in which a dialect i reaches the end of a conversation (i.e., sends a `DATA` or `QUIT` command, or just closes the connection), we assign a *gain* value g_i to the dialect i in that state. The gain value represents how much it would help achieve our detection goal if we reached that state during our decision process. Then, we propagate the gain values backwards along the transitions of the decision state machine. For each state s, we set the gain for i in that state as the maximum of the gain values for i that have been propagated to that state. To correctly handle loops, we continue propagating the gain values until we reach a fixed point. We then calculate the gain for s as the minimum of the gains for any dialect j in s. We do this to ensure that our decision is safe in the worst-case scenario (i.e., for the client with the minimal gain for that state). We calculate the initial gain for a state in different ways, depending on the goal of our decision process.

When performing spam mitigation, we want to avoid a legitimate client from failing to send an email. For this reason, we strongly penalize failure states for legitimate clients, while we want to have high gains for states in which spambots would fail. For each state in which a dialect reaches a final state, we calculate the gain for that state as follows: First, we assign a score to each client with a final label for that state (i.e., a `QUIT`, a `DATA`, or a connection closed label). We want to give more importance to states that make bots fail, while we never want to visit states that make legitimate clients fail. Also, we want to give a neutral gain to states that make legitimate clients succeed, and a slightly lower gain to states that make bots succeed. To achieve this, we assign a score of 1 for bot failure states, a score of 0 for legitimate clients failure states, a score of 0.5 for legitimate-client success states, and a score of 0.2 for bot success states. Notice that what we need here is a lattice of values that respect the stated precedence; therefore, any set of numbers that maintain this relationship would work.

When performing classification, we want to be as aggressive as possible in reducing the number of possible dialect candidates. In other words, we want to have high gains for states that allow us to make a decision on which dialect is spoken by a given client. Such states are those with a single possible client in them, or with different clients, each one with a different command label. To achieve this property, we set the gain for each state that includes a final label as $G = \frac{d}{n}$, where n is the total number of labels in that state, and d is the number of unique labels.

Reply selection. At each iteration of the algorithm explained in Section 5.2, we decide which reply to send by evaluating the gain for every possible reply from the states in A. For all the states reachable in one transition from the states in A, we first select the states S_a that still have at least an active client in their label table. We group together those states in S_a that are connected to the active states by transitions with the same label. For each label group, we pick the minimum gain among the states in that group. We consider this number as the gain we would get by sending that reply. After calculating the gain for all possible replies, we send the reply that has the highest gain associated to it. In case more than one reply yields the same gain we pick one randomly.

6 The Botnet Feedback Mechanism

Modern spamming botnets typically use *template-based spamming* to send out emails [22, 31, 38]. With this technique, the botnet C&C infrastructure tells the bots what kind of emails to send out, and the bots relay back information about the delivery as they received it from the SMTP server. This server feedback is an important piece of information to the botmaster, since it enables him to monitor if his botnet is working correctly.

Of course, a legitimate sender is also interested in information about the delivery process. However, she is interested in different information compared to the botmaster. In particular, a legitimate user wants to know whether the delivery of her emails failed (e.g., due to a typo in the email address). In such a case, the user wants to correct the mistake and send the message again. In contrast, a spammer usually sends emails in batches, and typically does not care about sending an email again in case of failure.

Nonetheless, there are three main pieces of information related to server feedback that a rational spammer is interested in: (i) whether the delivery failed because the IP address of the bot is blacklisted; (ii) whether the delivery failed because of specific policies in place at the receiving end (e.g., greylisting); (iii) whether the delivery failed because the recipient address does not exist. In all three cases, the spammer can leverage the information obtained from the mail server to make his operation more effective and profitable. In the case of a blacklisted bot, he can stop sending spam using that IP address, and wait for it to be whitelisted again after several hours or days. Empirical evidence suggests that spammers already collect this information and act accordingly [38]. If the recipient server replied with an SMTP non-critical error (i.e., the ones used in greylisting), the spammer can send the email again after some minutes to comply with the recipient's policy.

The third case, in which the recipient address does not exist, is the most interesting, because it implies that the spammer can *permanently* remove that email address from his email lists, and avoid using it during subsequent campaigns. Recent research suggests that bot feedback is an important part of a spamming botnet operation. For example, Stone-Gross et al. [38] showed that about 35% of the email addresses used by the *Cutwail* botnet were in fact non-existent. By leveraging the server feedback received by the bots, a rational botmaster can get rid of those non-existing addresses, and optimize his spamming performance significantly.

Breaking the Loop: Providing False Responses to Spam Emails. Based on these insights, we want to study how we can manipulate the SMTP delivery process of bots to influence their sending behavior. We want to investigate what would happen if mail servers started giving erroneous feedback to bots. In particular, we are interested in the third case, since influencing the first two pieces of information has only a limited, short-term impact on a spammer. However, if we provide false information about the status of a recipient's address, this leads to a double bind for the spammer: on the one hand, if a spammer considers server feedback, he will remove a valid recipient address from his email list. Effectively, this leads to a reduced number of spam emails received at this particular address. On the other hand, if the spammer does not consider server feedback, this reduces the effectiveness of his spam campaigns since emails are sent to non-existent addresses. In the long run, this will significantly degrade the freshness of his email lists and reduce the number of successfully sent emails. In the following, we discuss how we can take advantage of this situation.

As a first step, we need to identify that a given SMTP conversation belongs to a bot. To this end, a mail server can either use traditional, IP-based blacklists or leverage the analysis of SMTP dialects introduced previously. Once we have identified a bot, a mail server can (instead of closing the connection) start sending erroneous feedback to the bot, which will relay this information to the C&C infrastructure. Specifically, the mail server could, for example, report that the recipient of that email does not exist. By doing this, the email server would lead the botmaster to the lose-lose situation discussed before. For a rational botmaster, we expect that this technique would reduce the amount of spam the email address receives. We have implemented this approach as a second instance of our technique to leverage the email delivery for spam mitigation and report on the empirical results in Section 7.3.

7 Evaluation

In this section, we evaluate the effectiveness of our approach. First, we describe our analysis environment. Then, we evaluate both the dialects and the feedback manipulation techniques. Finally, we analyze the limitations and the possible evasion techniques against our system.

7.1 Analysis Environment

We implemented our approach in a tool, called `B@bel`. `B@bel` runs email clients (legitimate or malicious) in virtual machines, and applies the learning techniques explained in Section 4 to learn the SMTP dialect of each client. Then, it leverages the learned dialects to build a decision machine M_D, and uses it to perform malware classification or spam mitigation.

The first component of `B@bel` is a virtual machine zoo. Each of the virtual machines in the zoo runs a different *email client* [4]. Clients can be legitimate email programs, mail transfer agents, or spambots.

The second component of `B@bel` is a gateway, used to confine suspicious network traffic. Since the clients that we run in the virtual machines are potentially malicious, we need to make sure that they do not harm the outside world. To this end, while still allowing the clients to connect to the Internet, we use restricting firewall rules, and we throttle their bandwidth, to make sure that they will not be able to launch denial of service attacks. Furthermore, we sinkhole all SMTP connections, redirecting them to local mail servers under our control.

We use three different mail servers in `B@bel`. The first email server is a regular server that speaks plain SMTP, and will perform passive observation of the client's SMTP conversation. The second server is instrumented to perform active probing, as described in Section 4.2. Finally, the third server is configured to always report to the client that the recipient of an email does not exist, and is used to study how spammers use the feedback they receive from their bots.

The third component of `B@bel` is the learner. This component analyzes the active or passive observations generated between the clients in the zoo and the mail servers, learns an SMTP dialect for each client, and generates the decision state machine using the various dialects as input, as explained in Section 5. According to the task we want to perform (dialect classification or spam mitigation), `B@bel` tags the states in the decision state machine with the appropriate gain.

The last component of `B@bel` is the decision maker. This component analyzes an SMTP conversation, by either passively observing it or by impersonating the server, and makes a decision about which dialect is spoken by the client, using the process described in Section 5.2.

7.2 Evaluating the Dialects

Evaluating Dialects for Classification We trained `B@bel` by running active probing on a variety of popular Mail User Agents, Mail Transfer Agents, and bot samples. Table 1 lists the clients we used for dialect learning. Since we are extracting dialects by looking at the SMTP conversations only, `B@bel` is agnostic to the family a bot belongs to. However, for legibility purposes, Table 1 groups bots according to the most frequent label assigned by the anti-virus products deployed by VirusTotal [44]. Our dataset contained 13 legitimate MUAs and MTAs, and 91 distinct malware samples[5]. We picked the spambot samples to be representative of the largest active spamming botnets according to a recent report [26] (the report lists *Lethic*, *Cutwail*, *Mazben*, *Cutwail*, *Tedroo*, *Bagle*). We also picked worm samples that spread through email, such as *Mydoom*. In total, the malware samples we selected belonged to 11 families. The dialect learning phase resulted in a total of 60 dialects. We explain the reason for the high number of discovered dialects later in this section.

We then wanted to assess whether a dialect (i.e., a state machine) is unique or not. For each combination of dialects $<d_1, d_2>$, we merged their state machines together as explained in Section 5.1. We consider two dialects as distinct if any state of the merged state machine has two different labels in the label table for the dialects d_1 and d_2, or if any state has a single possible dialect in it.

The results show that the dialects spoken by the legitimate MUAs and MTAs are distinct from the ones spo-

[4]We used VirtualBox as our virtualization environment, and Windows XP SP3, Windows Server 2008, Windows 7, Ubuntu Linux 11.10, or Mac OS X Lion as operating systems on the virtual machines, depending on the operating system needed to run each of the legitimate clients or MTAs. We used Windows XP SP3 to run the malware samples

[5]The MD5 checksums of the malware samples are available at `http://cs.ucsb.edu/~gianluca/files/babel.txt`

Mail User Agents	**Mail Transfer Agents**	**Bots (by AV labels)**
Eudora, Opera, Outlook 2010, Outlook Express, Pegasus, The Bat!, Thunderbird, Windows Live Mail	Exchange 2010, Exim, Postfix, Qmail, Sendmail	Waledac, Donbot, Grum, Klez Buzus, Bagle, Lethic, Cutwail, Mydoom, Mazben, Tedroo

Table 1: MTAs, MUAs, and bots used to learn dialects.

ken by the bots. By analyzing the set of dialects spoken by legitimate MUAs and MTAs, we found that they all speak distinct dialects, except for Outlook Express and Windows Live Mail. We believe that Microsoft used the same email engine for these two products.

The 91 malware samples resulted in 48 unique dialects. We manually analyzed the spambots that use the same dialect, and we found that they always belong to the same family, with the exception of six samples. These samples were either not flagged by any anti-virus at the time of our analysis, or match a dropper that downloaded the spambot at a later time [8]. This shows that `B@bel` is able to classify spambot samples by looking at their email behavior, and label them more accurately than anti-virus products.

We then wanted to understand the reason for the high number of dialects we discovered. To this end, we considered clusters of malware samples that were talking the same dialect. For each cluster, we assigned a label to it, based on the most common anti-virus label among the samples in the cluster. All the clusters were unique, with the exception of eleven clusters marked as Lethic and two clusters marked as Mydoom. By manual inspection, we found that Lethic randomly closes the connection after sending the `EHLO` message. Since our dialect state machines are nondeterministic, our approach handles this case, in principle. However, in some cases, this nondeterministic behavior made it impossible to record a reply for a particular test case during our active probing. We found that each cluster labeled as Lethic differs for at most five non-recorded test cases with every other Lethic cluster. This gives us confidence to say that the dialect spoken by Lethic is indeed unique. For the two clusters labeled as Mydoom, we believe this is a common label assigned to unknown worms. In fact, the two dialects spoken by the samples in the clusters are very different. This is another indicator that `B@bel` can be used to classify spamming malware in a more precise fashion than is possible by relying on anti-virus labels only.

Evaluating Dialects for Spam Detection To evaluate how the learned dialects can be used for spam detection, we collected the SMTP conversations for 621,919 email messages on four mail servers in our department, spanning 40 days of activity.

For each email received by the department servers, we extracted the SMTP conversation associated with it, and then ran `B@bel` on it to perform spam detection. To this end, we used the conversations logged by the *Anubis* system [4] during a period of one year (corresponding to 7,114 samples) to build the bot dialects, and the dialects learned in Section 7.2 for MUAs and MTAs as legitimate clients. In addition, we manually extracted the dialects spoken by popular web mail services from the conversations logged by our department mail servers, and added them to the legitimate MTAs dialects. Note that, since the goal of this experiment is to perform passive spam detection, learning the dialects by passively observing SMTP conversations is sufficient.

During our experiment, `B@bel` marked any conversation as spam if, at the end of the conversation, the dialects in C_D were all associated with bots. Furthermore, if the dialects in C_D were all associated with MUAs or MTAs, `B@bel` marked the conversation as legitimate (ham). If there were both good and malicious clients in C_D, `B@bel` did not make a decision. Finally, if the decision state machine did not recognize the SMTP conversation at all, `B@bel` considered that conversation as spam. This could happen when we observe a conversation from a client that was not in our training set. As we will show later, considering it as spam is a reasonable assumption, and is not a major source of false positives.

In total, `B@bel` flagged 260,074 conversations as spam, and 218,675 as ham. For 143,170 emails, `B@bel` could not make a decision, because the decision process ended up in a state where there were both legitimate clients and bots in C_D.

To verify how accurate our decisions were, we used a number of techniques. First, we checked whether the email was blocked by the department mail servers in the first place. These servers have a common configuration, where incoming emails are first checked against an IP blacklist, and then against more expensive content-analysis techniques. In particular, these servers used a commercial blacklist for discarding emails coming from known spamming IP addresses, and SpamAssassin and ClamAV for content analysis. Any time one of these techniques and `B@bel` agreed on flagging a conversation as spam, we consider this as a true positive of our system. We also consider as a true positive those conversations `B@bel` marked as spam, and that lead to an `NXDOMAIN` or to a timeout when we tried to resolve the domain associated to the sender email address. In addition, we checked the sender IP address against 30 additional IP blacklists[6], and considered any match as a true positive. According to this ground truth, the true positive rate for the emails B@bel flagged as being sent by bots is 99.32%. Surprisingly, 98% of the 24,757 conversations that were not recognized by our decision state machine were flagged as spam by existing methods. This shows that, even if the set of clients from which B@bel learned the dialects from is not complete, there are no widely-used legitimate clients we missed, and that it is safe to consider any conversation generated by a non-observed dialect as spam. For the remaining 2,074 emails that B@bel flagged as spam, we could not assess if they were spam or not. They might have been a false positive of B@bel, or a false negative of the existing methods. To remain on the safe side, we consider them as false positives. This results in B@bel having a precision of 99.3%.

We then looked at our false negatives. We consider as false negatives those conversations that B@bel classified as belonging to a legitimate client dialect, but that have been flagged as spam by any of the previously mentioned techniques. In total, the other spam detection mechanisms flagged 71,342 emails as spam, among the ones that B@bel flagged as legitimate. Considering these emails as false negatives, this results in B@bel having a false negative rate of 21%. The number of false negatives might appear large at first. However, we need to consider the sources of these spam messages. While the vast majority of spam comes from botnets, spam can also be sent by dedicated MTAs, as well as through misused web mail accounts. Since B@bel is designed to detect email clients, we are able to detect which MTA or web mail application the email comes from, but we cannot assess whether that email is ham or spam. To show that this is the case, we investigated these 71,342 messages, which originated from 7,041 unique IP addresses. Assuming these are legitimate MTAs, we connected to each IP address on TCP port 25 and observed greeting messages for popular MTAs. For 3,183 IP addresses, one of the MTAs that we used to learn the dialects responded. The remaining 3,858 IP addresses did not respond within a 10 second timeout. We performed reverse DNS lookups on these IP addresses and assessed whether their assigned DNS names contained indicative names such as `smtp` or `mail`. 1,654 DNS names were in this group. We could not find any conclusive proof that the remaining 2,204 addresses belong to legitimate MTAs.

For those dialects for which B@bel could not make a decision (because the conversation lead to a state where both one or more legitimate clients and bots were active), we investigated if we could have assessed whether the client was a bot or not by using active probing. Since the spambot and legitimate client dialects that we observed are disjoint, this is always possible. In particular, B@bel found that it is always possible to distinguish between the dialects spoken by a spambot and by a legitimate email client that look identical from passive analysis by sending a single SMTP reply. For example, the SMTP RFC specifies that multi-line replies are allowed, in the case all the lines in the reply have the same code, and all the reply codes but the last one are followed by a dash character. Therefore, multi-line replies that use different reply codes are not allowed by the standard. We can leverage different handling of this corner case to disambiguate between Qmail and Mydoom. More precisely, if we send the reply `250-OK<CR><LF>550 Error`, Qmail will take the first reply code as the right one, and continue the SMTP transaction, while Mydoom will take the second reply code as the right one, and close the connection. Based on these observations, we can say that if we ran B@bel in active mode, we could distinguish between these ambiguous cases, and make the right decision. Unfortunately, we could run B@bel only in passive mode on our department mail servers.

Our results show that B@bel can detect (and possibly block) spam emails sent by bots with high accuracy. However, B@bel is unable to detect those spam emails sent by dedicated MTAs or by compromised webmail accounts. For this reason, similar to the other state-of-the-art mitigation techniques, B@bel is not a silver bullet, but should be used in combination with other anti-spam mechanisms. To show what would be the advantage of deploying B@bel on a mail server, we studied how much spam would have been blocked on our department server if B@bel was used in addition to or in substitution to the commercial blacklist and the content analysis systems that are currently in use on those servers.

Similarly to IP blacklists, B@bel is a lightweight technique. Such techniques are typically used as a first spam-mitigation step to make quick decisions, as they avoid having to apply resource-intensive content analysis techniques to most emails. For this reason, the first configuration we studied is substituting the commercial blacklist with B@bel. In this case, 259,974 emails would have been dropped as spam, instead of the 219,726 that were blocked by the IP blacklist. This would have resulted in 15.5% less emails being sent to the content analysis system, reducing the load on the servers. Moreover, the emails detected as spam by B@bel and the IP blacklist do not overlap completely. For example, the IP blacklist flags as spam emails sent by known misused MTAs. Therefore, we analyzed the amount of spam that the two techniques could have caught if used together. In this scenario, 278,664 emails would have been blocked,

[6]The blacklists we leveraged come from these services: Barracuda, CBL, Spamhaus, Atma, Spamcop, Manitu, AHBL, DroneBL, DShield, Emerging Threats, malc0de, McAfee, mdl, OpenBL, SORBS, Sucuri Security, TrendMicro, UCEPROTECT, and ZeusTracker. Note that some services provide multiple blacklists

resulting in 26.8% less emails being forwarded to the content analysis system compared to using the blacklist alone. As a last experiment, we studied how much spam would have been blocked on our servers by using `B@bel` in combination with both the commercial blacklist and the content analysis systems. In this scenario, 297,595 emails would have been flagged as spam, which constitutes an improvement of 3.9% compared to the servers' original configuration.

7.3 Evaluating the Feedback Manipulation

To investigate the effects of wrong server feedback to bots, we set up the following experiment. We ran 32 malware samples from four large spamming botnet families (*Cutwail*, *Lethic*, *Grum*, and *Bagle*) in a controlled environment, and redirected all of their SMTP activity to the third mail server in the `B@bel` architecture. We configured this server to report that *any* recipient of the emails the bots were sending to was non-existent, as described in Section 7.1.

To assess whether the different botnets stopped sending emails to those addresses, we leveraged a *spamtrap* under our control. A spamtrap is a set of email addresses that do not belong to real users, and, therefore, collect only spam mails. To evaluate our approach, we leverage the following idea: if an email address is successfully removed from an email list used by a spam campaign, we will not observe the same campaign targeting that address again. We define as a spam campaign the set of emails that share the same URL templates in their links, similar to the work of Xie et al. [48]. While there are more advanced methods to detect spam campaigns [31], the chosen approach leads to sufficiently good results for our purposes.

We ran our experiment for 73 days, from June 18 to August 30, 2011. During this period, our mail server replied with false server feedback for 3,632 destination email addresses covered by our spamtrap, which were targeted by 29 distinct spam campaigns. We call the set of campaigns C_f and the set of email addresses S_f. Of these, five campaigns never targeted the addresses for which we gave erroneous feedback again. To estimate the probability P_c that the spammer running campaign c in C_f actually removed the addresses from his list, and that our observation is not random, we use the following formula:

$$P_c = 1 - (1 - \tfrac{n}{t_f - t_b})^{t_e - t_f},$$

where n is the total number of emails received by S_f for c, t_f is the time at which we first gave a negative feedback for an email address targeted by c, t_b is the first email for c which we ever observed targeting our spam trap, and t_e is the last email we observed for c. This formula calculates the probability that, given a certain number n of emails observed for a certain campaign c, no email was sent to the email addresses in S_f *after* we sent a poisoned feedback for them. We calculate P_c for the five campaigns mentioned above. For three of them, the confidence was above 0.99. For the remaining two, we did not observe enough emails in our spamtrap to be able to make a final estimate.

To assess the impact we would have had when sending erroneous feedback to all the addresses in the spamtrap, we look at how many emails the whole spamtrap received from the campaigns in C_f. In total, 2,864,474 emails belonged to campaigns in C_f. Of these, 550,776 belonged to the three campaigns for which we are confident that our technique works and reduced the amount of spam emails received at these addresses. Surprisingly, this accounts for 19% of the total number of emails received, indicating that this approach could have impact in practice.

We acknowledge that these results are preliminary and provide only a first insight into the large-scale application of server feedback poisoning. Nevertheless, we are confident that this approach is reasonable since it leads to a lose-lose situation for the botmaster, as discussed in Section 6. We argue that the uncertainty about server feedback introduced by our method is beneficial since it reduces the amount of information a spammer can obtain when sending spam.

7.4 Limitations and Evasion

Our results demonstrate that `B@bel` is successful in detecting current spambots. However, spam detection is an adversarial game. Thus, once `B@bel` is deployed, we have to expect that spammers will evolve and try to bypass our systems. In this section, we discuss potential paths for evasion.

Evading dialects detection. The most immediate path to avoid detection by dialects is to implement an SMTP engine that precisely follows the specification. Alternatively, a bot author could make use of an existing (open source) SMTP engine that is used by legitimate email clients. We argue that this has a negative impact on the effectiveness and flexibility of spamming botnets.

Many spambots are built for performance; their aim is to distribute as many messages as possible. In some cases, spambots even send multiple messages without waiting for any server response. Clearly, any additional checks and parsing of server replies incurs overhead that might slow down the sender. We performed a simple experiment to measure the speed difference between a malware program sending spam (`Bagle`) and a legitimate email library on Windows (`Collaboration Data Objects - CDO`). We found that Bagle can send an email every 20 ms to a local mail server. When trying to send emails as fast as possible using the Windows library (in a tight loop), we measured that a single email required 200 ms, an order of magnitude longer. Thus, when bots are forced to faithfully implement large portions of the SMTP specification (because otherwise, active probing will detect differences), spammers suffer a performance penalty.

Spammers could still decide to adopt a well-known SMTP implementation for their bots, run a full, parallelized, SMTP implementation, or revert to a well-known SMTP library when they detect that the recipient server is using `B@bel` for detection. In this case, another aspect of spamming botnets has to be taken into account. Typically, cyber criminals who infect machines with bots are not the same as the spammers who rent botnets to distribute their messages. Modern spamming botnets allow their customers to customize the email headers to mimic legitimate clients. In this scenario, `B@bel` could exploit possible discrepancies between the email client identified by the SMTP dialect and the one announced in the body of an email (for example, via the `X-Mailer` header). When these two dialects do not match (and the SMTP dialect does not indicate an MTA), we can detect that the sender pretends to speak a dialect that is inconsistent with the content of the (spam) message. Of course, the botmasters could take away the possibility for their customers to customize the headers of their emails, and force them to match the ones typical of a certain legitimate client (e.g., Outlook Express). However, while this would make spam detection harder for `B@bel`, it would make it easier for other systems that rely on email-header analysis, such as *Botnet Judo* [31], because spammers would be less flexible in the way they vary their templates.

Mitigating feedback manipulation. As we discussed in Section 6, spammers can decide to either discard any feedback they receive from the bots, or trust this feedback. To avoid this, attackers could guess whether the receiving mail server is performing feedback manipulation. For example, when all emails to a particular domain are rejected because no recipient exists, maybe all feedback from this server can be discarded. In this case, we would need to update our feedback mechanism to return invalid feedback only in a fraction of the cases.

8 Related Work

Email spam is a well-known problem that has attracted a substantial amount of research over the past years. In the following, we briefly discuss how our approach is related to previous work in this area and elaborate on the novel aspects of our proposed methods.

Spam Filtering: Existing work on spam filtering can be broadly classified in two categories: *post-acceptance methods* and *pre-acceptance methods*. Post-acceptance methods receive the full message and then rely on *content analysis* to detect spam emails. There are many approaches that allow one to differentiate between spam and legitimate emails: popular methods include Naive Bayes, Support Vector Machines (SVMs), or similar methods from the field of machine learning [16, 27, 35, 36]. Other approaches for content-based filtering rely on identifying the URLs used in spam emails [2,48]. A third method is *DomainKeys Identified Mail* (DKIM), a system that verifies that an email has been sent by a certain domain by using cryptographic signatures [23]. In practice, performing content analysis or computing cryptographic checksums on every incoming email can be expensive and might lead to high load on busy servers [41]. Furthermore, an attacker might attempt to bypass the content analysis system by crafting spam messages in specific ways [25, 28]. In general, the drawback of post-acceptance methods is that an email has to be received before it can be analyzed.

Pre-acceptance methods attempt to detect spam before actually receiving the full message. Some analysis techniques take the *origin* of an email into account and analyze distinctive features about the sender of an email (e.g., the IP address or autonomous system the email is sent from, or the geographical distance between the sender and the receiver) [17,34,39,43]. In practice, these sender-based techniques have coverage problems: previous work showed how IP blacklists miss detecting a large fraction of the IP addresses that are actually sending spam, especially due to the highly dynamic nature of the machines that send spam (typically botnets) [32, 37, 38].

Our method is a novel, third approach that focuses on *how* messages are sent. This avoids costly content analysis, and does not require the design and implementation of a reputation metric or blacklist. In contrast, we attempt to recognize the SMTP dialect during the actual SMTP transaction, and our empirical results show that this approach can successfully discriminate between spam and ham emails. This complements both pre-acceptance and post-acceptance approaches. Another work that went in this direction was done by Beverly et al. [5] and Kakavelakis et al. [19]. The authors of these two papers leveraged the fact that spambots have often bad connections to the Internet, and perform spam detection by looking at TCP-level features such as retransmissions and connection resets. Our system is more robust, because it does not rely on assumptions based on the network connectivity of a mail client.

Moreover, to the best of our knowledge, we are the first to study the effects of manipulating server feedback to poison the information sent by a bot to the botmaster.

Protocol Analysis: The core idea behind our approach is to learn the SMTP dialect spoken by a particular client. This problem is closely related to the problem of automated protocol reverse-engineering, where an (unknown) protocol is analyzed to determine the individual records/elements and the protocol's structure [6,13]. Initial work in this area focused on clustering of network traces to group similar messages [14], while later methods extracted protocol information by analyzing the execution of a program while it performs network communication [10,15,24,45,47]. Sophisticated methods can also handle multiple messages and recover the protocol's state machine. For example, *Dispatcher* is a tool capable of extracting the format of protocol messages when having access to only one endpoint, namely the bot binary [9]. Cho et al. leverage the information extracted by Dispatcher to learn C&C protocols [11]. Brumley et al. studied how deviations in the implementation of a given protocol specification can be used to detect errors or generate fingerprints [7]. The differences in how a given program checks and processes inputs are identified with the help of binary analysis (more specifically, symbolic execution).

Our problem is related to previous work on protocol analysis, in the sense that we extract different SMTP protocol variations, and use these variations to build fingerprints. However, in this work, we treat the speaker of the protocol (the bot) as a blackbox, and we do not perform any code analysis or instrumentation to find protocol formats or deviations. This is important because (i) malware is notoriously difficult to analyze and (ii) we might not always have a malware sample available. Instead, our technique allows us to build SMTP dialect state machines even when interacting with a previously-unknown spambot.

There is also a line of research on fingerprinting protocols [12,30,49]. Initial work in this area leveraged manual analysis. Nonetheless, there are methods, such as *FiG*, that automatically generate fingerprints for DNS servers [42]. The main difference between our work and *FiG* is that our dialects are stateful while *FiG* operates on individual messages. This entirely avoids the need to merge and explore protocol state machines. However, as discussed previously, individual messages are typically not sufficient to distinguish between SMTP engines.

9 Conclusion

In this paper, we introduced a novel way to detect and mitigate spam emails that complements content- and sender-based analysis methods. We focus on *how* email messages are sent and derive methods to influence the spam delivery mechanism during SMTP transactions. On the one hand, we show how small deviations in the SMTP implementation of different email agents (so called *SMTP dialects*) allow us to detect spambots during the actual SMTP communication. On the other hand, we study how the feedback mechanism used by botnets can be poisoned, which can be used to have a negative impact on the effectiveness of botnets.

Empirical results confirm that both aspects of our approach can be used to detect and mitigate spam emails. While spammers might adapt their spam-sending practices as a result of our findings, we argue that this reduces their performance and flexibility.

Acknowledgments

This work was supported by the Office of Naval Research (ONR) under Grant N000140911042, the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, by Secure Business Austria, and by the German Federal Ministry of Education and Research under grant 01BY1111 / MoBE. We want to thank our shepherd Alex Moshchuk and the anonymous reviewers for their valuable comments, and Andreas Boschke for his help in setting up some of our experiments.

References

[1] RFC 821: Simple Mail Transfer Protocol. `http://tools.ietf.org/html/rfc821`.

[2] SURBL URI reputation data. `http://www.surbl.org/`.

[3] The Spamhaus Project. `http://www.spamhaus.org`.

[4] BAYER, U., MOSER, A., KRUEGEL, C., AND KIRDA, E. Dynamic analysis of malicious code. *Journal in Computer Virology 2*, 1 (2006), 67–77.

[5] BEVERLY, R., AND SOLLINS, K. Exploiting Trasport-level Characteristics of Spam. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2008).

[6] BORISOV, N., BRUMLEY, D., WANG, H. J., DUNAGAN, J., JOSHI, P., AND GUO, C. Generic Application-Level Protocol Analyzer and its Language. In *Symposium on Network and Distributed System Security (NDSS)* (2007).

[7] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOM, J., AND SONG, D. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *USENIX Security Symposium* (2007).

[8] CABALLERO, J., GRIER, C., KREIBICH, C., AND PAXSON, V. Measuring Pay-per-Install: The Commoditization of Malware Distribution. In *USENIX Security Symposium* (2011).

[9] CABALLERO, J., POOSANKAM, P., KREIBICH, C., AND SONG, D. X. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[10] CABALLERO, J., YIN, H., LIANG, Z., AND SONG, D. X. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[11] CHO, C. BABIC, D. S. D. Inference and Analysis of Formal Models of Botnet Command and Control Protocols. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[12] COMER, D. E., AND LIN, J. C. Probing TCP Implementations. In *USENIX Summer Technical Conference* (1994).

[13] COMPARETTI, P. M., WONDRACEK, G., KRUEGEL, C., AND KIRDA, E. Prospex: Protocol Specification Extraction. In *IEEE Symposium on Security and Privacy* (2009).

[14] CUI, W., KANNAN, J., AND WANG, H. J. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In *USENIX Security Symposium* (2007).

[15] Cui, W., Peinado, M., Chen, K., Wang, H. J., and Irunbriz, L. Tupni: automatic reverse engineering of input formats. In *ACM Conference on Computer and Communications Security (CCS)* (2008).

[16] Drucker, H., Wu, D., and Vapnik, V. N. Support vector machines for spam categorization. In *IEEE transactions on neural networks* (1999).

[17] Hao, S., Syed, N. A., Feamster, N., Gray, A. G., and Krasser, S. Detecting Spammers with SNARE: Spatio-temporal Network-level Automatic Reputation Engine. In *USENIX Security Symposium* (2009).

[18] John, J. P., Moshchuk, A., Gribble, S. D., and Krishnamurthy, A. Studying Spamming Botnets Using Botlab. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2009).

[19] Kakavelakis, G., Beverly, R., and J., Y. Auto-learning of SMTP TCP Transport-Layer Features for Spam and Abusive Message Detection. In *USENIX Large Installation System Administration Conference* (2011).

[20] Kanich, C., Weaver, N., McCoy, D., Halvorson, T., Kreibich, C., Levchenko, K., Paxson, V., Voelker, G., and Savage, S. Show Me the Money: Characterizing Spam-advertised Revenue. *USENIX Security Symposium* (2011).

[21] Kaspersky Lab. Spam Report: April 2012. `https://www.securelist.com/en/analysis/204792230/Spam_Report_April_2012`, 2012.

[22] Kreibich, C., Kanich, C., Levchenko, K., Enright, B., Voelker, G. M., Paxson, V., and Savage, S. On the Spam Campaign Trail. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2008).

[23] Leiba, B. DomainKeys Identified Mail (DKIM): Using digital signatures for domain verification. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2007).

[24] Lin, Z., Jiang, X., Xu, D., and Zhang, X. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *Symposium on Network and Distributed System Security (NDSS)* (2008).

[25] Lowd, D., and Meek, C. Good word attacks on statistical spam filters. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2005).

[26] M86 Labs. Security labs report. `http://www.m86security.com/documents/pdfs/security_labs/m86_security_labs_report_2h2011.pdf`, 2011.

[27] Meyer, T., and Whateley, B. SpamBayes: Effective open-source, Bayesian based, email classification system. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2004).

[28] Nelson, B., Barreno, M., Chi, F. J., Joseph, A. D., Rubinstein, B. I. P., Saini, U., Sutton, C., Tygar, J. D., and Xia, K. Exploiting Machine Learning to Subvert Your Spam Filter. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2008).

[29] Pathak, A., Hu, Y. C., and Mao, Z. M. Peeking into spammer behavior from a unique vantage point. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2008).

[30] Paxson, V. Automated Packet Trace Analysis of TCP Implementations. In *ACM SIGCOMM Conference* (1997).

[31] Pitsillidis, A., Levchenko, K., Kreibich, C., Kanich, C., Voelker, G. M., Paxson, V., Weaver, N., and Savage, S. botnet Judo: Fighting Spam with Itself. In *Symposium on Network and Distributed System Security (NDSS)* (2010).

[32] Ramachandran, A., Dagon, D., and Feamster, N. Can DNS-based blacklists keep up with bots? In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2006).

[33] Ramachandran, A., and Feamster, N. Understanding the Network-level Behavior of Spammers. *SIGCOMM Comput. Commun. Rev. 36* (August 2006).

[34] Ramachandran, A., Feamster, N., and Vempala, S. Filtering Spam with Behavioral Blacklisting. In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[35] Sahami, M., Dumais, S., Heckermann, D., and Horvitz, E. A Bayesian approach to filtering junk e-mail. *Learning for Text Categorization* (1998).

[36] Sculley, D., and Wachman, G. M. Relaxed Online SVMs for Spam Filtering. In *ACM SIGIR Conference on Research and Development in Information Retrieval* (2007).

[37] Sinha, S., Bailey, M., and Jahanian, F. Shades of Grey: On the Effectiveness of Reputation-based "Blacklists". In *International Conference on Malicious and Unwanted Software* (2008).

[38] Stone-Gross, B., Holz, T., Stringhini, G., and Vigna, G. The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-Scale Spam Campaigns. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2011).

[39] Stringhini, G., Holz, T., Stone-Gross, B., Kruegel, C., and Vigna, G. BotMagnifier: Locating Spammers on the Internet. In *USENIX Security Symposium* (2011).

[40] Symantec Corp. State of spam & phishing report. `http://www.symantec.com/business/theme.jsp?themeid=state_of_spam`, 2010.

[41] Taylor, B. Sender reputation in a large webmail service. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2006).

[42] Venkataraman, S., Caballero, J., Poosankam, P., Kang, M. G., and Song, D. X. FiG: Automatic Fingerprint Generation. In *Symposium on Network and Distributed System Security (NDSS)* (2007).

[43] Venkataraman, S., Sen, S., Spatscheck, O., Haffner, P., and Song, D. Exploiting Network Structure for Proactive Spam Mitigation. In *USENIX Security Symposium* (2007).

[44] VirusTotal. Free Online Virus, Malware and URL Scanner. `https://www.virustotal.com/`.

[45] Wang, Z., Jiang, X., Cui, W., Wang, X., and Grace, M. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *European Symposium on Research in Computer Security (ESORICS)* (2009).

[46] Wolf, W. An Algorithm for Nearly-Minimal Collapsing of Finite-State Machine Networks. In *IEEE International Conference on Computer-Aided Design (ICCAD)* (1990).

[47] Wondracek, G., Comparetti, P. M., Kruegel, C., and Kirda, E. Automatic Network Protocol Analysis. In *Symposium on Network and Distributed System Security (NDSS)* (2008).

[48] Xie, Y., Yu, F., Achan, K., Panigrahy, R., Hulten, G., and Osipkov, I. Spamming Botnets: Signatures and Characteristics. *SIGCOMM Comput. Commun. Rev. 38* (August 2008).

[49] Zalewski, M. p0f v3. `http://lcamtuf.coredump.cx/p0f3/`, 2012.

[50] Zhuang, L., Dunagan, J., Simon, D. R., Wang, H. J., and Tygar, J. D. Characterizing Botnets From Email Spam Records. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2008).