# Automatic Adaptation and Analysis of SIP Headers Using Decision Trees

Andrea Hess and Michael Nussbaumer and Helmut Hlavacs and Karin Anna Hummel

Department of Distributed and Multimedia Systems
University of Vienna, Austria
Lenaugasse 2/8, A-1080 Vienna
http://www.cs.univie.ac.at/
andrea.hess|karin.hummel|helmut.hlavacs@univie.ac.at,
michael.nussbaumer@ani.univie.ac.at

**Abstract.** Software implementing open standards like SIP evolves over time, and often during the first years of deployment, products are either immature or do not implement the whole standard but rather only a subset. As a result, messages compliant to the standard are sometimes wrongly rejected and communication fails. In this paper we describe a novel approach called Babel-SIP for increasing the rate of acceptance for SIP messages.
Babel-SIP is a filter that is put in front of a SIP parser and analyzes incoming SIP messages. It gradually learns which messages are likely to be accepted by the parser, and which are not. Those classified as probably rejected are then adapted such that the probability for acceptance is increased. In a number of experiments we demonstrate that our filter is able to drastically increase the acceptance rate of problematic SIP REGISTER and INVITE messages. Additionally we show that our approach can be used to analyze the faulty behavior of a SIP parser by using the generated decision trees.

**Keywords:** Protocol adaptation, decision tree based learning, SIP

## 1 Introduction

One of the success factors of the Internet and of many of its applications is the openness of its protocols. Thousands of protocols are described in the form of request for comment (RFC), some being simple and described by one single RFC, some being spread over several RFCs, where each RFC might either describe one important aspect of the protocol, or even comprise a suit of closely related protocols rather than one single protocol. Due to the complexity of many protocols, it is often not possible to create suitable protocol stacks from scratch which implement the open standards completely and flawlessly right from the start. Rather, it is often better to first implement a subset of the most important features of a protocol, then before shipping the product, to try to run as many test runs as possible for debugging. After the first release, software producers then try to gradually implement missing features and continuously run debugging and compatibility tests. Since many companies develop their products like this, and also since many complex standards do leave some questions unanswered, it regularly may happen that devices adhering to the same protocol standard are unable to communicate with each other.

Consider SIP [1] for instance, which will be introduced in more detail in Section 3. SIP is a protocol for call session establishment and management, on which voice over IP (VoIP), for instance, is based. It is thus the glue binding together phones on the one side, and telephone infrastructure like proxy servers on the other side. Due to the multitude of products around SIP, in the recent years, many incompatibilities between phones and proxies have been observed (despite events like SIPit[1]). For testing compatibility, commercial VoIP proxy vendors usually purchase a set of hard and soft phones, and then test them against their product (of course many other software and conformance tests are run as well). Together with the proxy software, vendors then often specify a list of hard phones or soft phones which are known to work with their product. Proxy customers are in turn advised to use phones from this list. For instance, for the commercial proxy considered in this work, during the recent versions, several hard phones were known which would not be able to register themselves to the proxy. In case incompatibilities arise, proxy customers usually must wait until the proxy vendor acknowledges and removes the observed incompatibilities in the next patch or proxy release, something which might take weeks or even months.

In the long run, products get more and more robust, and compatibility issues are gradually removed. However, in the transient phase of initial deployment, usually during the first years of a newly proposed protocol, such incompatibilities may cause a lot of despair.

In this paper we present Babel-SIP, a novel SIP translator that is able to improve the situation significantly. Babel-SIP can be plugged in front of a proxy, and automatically analyzes incoming SIP messages. It gradually learns, which kind of SIP messages are likely to be accepted by the proxy SIP parser, and which are likely to be rejected because of the above described incompatibilities. The same learning concept can then be used to pro-actively adapt incoming SIP messages which are likely to cause trouble in such a way that the new version of the SIP message is likely to be accepted by the SIP parser.

We consider our approach to be generic in the sense that it is not necessarily restricted to be used for SIP. Rather, it is thinkable to construct other versions of Babel-SIP for newly proposed protocols in order to improve transient phases for newly deployed products.

[1] http://www.cs.columbia.edu/sip/sipit/

## 2 Related Work

Since protocols are either proprietary or standardized, using autonomous self-adapting parsers based on machine learning techniques is not as widespread as in other domains such as robotics, natural language classification, node and network utilization including estimates of future utilization, and intrusion detection.

Decision trees, and in particular the used C4.5 tree, allow to classify arbitrary entities or objects which can be used, for instance for computer vision (applied to robotics) [2] or characterization of computer resource usage [3]. In [2] decision trees were used for learning about the visual environment which was modeled in terms of simple and complex attributes and successfully implemented for improving recognition possibilities of Sony Aibo robots (e.g., the surface area or angles). Decision trees which use further linear regression have been proposed for the characterization of computer resource usage in [3]. Parameters like the CPU, I/O, and memory were used as attributes and the classification tree was finally used to successfully determine anomalies of the system's parameters. The authors claim that finding the trade-off between accurate history knowledge and time-consuming training was a major concern.

In [4] intrusion detection was introduced based on a combination of pattern matching and decision tree-based protocol analysis. This tree-based approach allows to adapt to new attack types and forms while the traditional patterns are integrated into the tree and benefit from refinement of crucial parameters. All presented decision tree based approaches are similar to our approach and are mentioned to motivate the potential for protocol analysis and message classification.

In the application area of VoIP and SIP, authors both investigate traffic behavior and failures in particular software implementations. In [5] the authors describe the need and their solution for profiling SIP-based VoIP traffic (protocol behavior) to automatically detect anomalies. They demonstrate that SIP traffic can be modeled well with their profiling and that anomalies could be detected. In [6] it is argued, that based on the SIP specification, a formal testing of an open source and a commercial SIP proxy lead to errors with the SIP registrar. Both findings are encouraging to propose a method for not only detecting incompatibilities and testing SIP proxies, but further to provide a solution for messages rejected due to slightly different interpretations of the standard or software faults.

In [7] a stateful fuzzer was used to test the SIP compatibility of User Agents and proxies by sending different (faulty) messages both in terms of syntax and in terms of protocol behavior. The idea here is only to find weaknesses in the parser implementation, without trying to adapt messages online. The work closest to our approach is presented in [8]. In this approach, incoming and outgoing SIP messages of a proxy are analyzed by an in-kernel Linux classification engine. Hereby, a rule-based approach is proposed, where the rules are pre-defined (static). Our approach extends this classification by proposing a generalizable solution capable of learning. Additionally, we propose the novel approach of autonomic adaptation and evaluate it.

## 3 Background: The Session Initiation Protocol

The Session Initiation Protocol (SIP) is gradually becoming a key protocol for many application areas, such as voice over IP, or general session management of the Internet Multimedia Subsystem (IMS) of Next Generation Networks (NGNs). Its functions target (i) user location, (ii) user availability, (iii) user capabilities, (iv) session setup, and (v) session management. The core SIP functionality is defined in RFC 3261 [1] but several other RFCs define different additional aspects of SIP, like reliability [9], interaction with DNS [10], events and notifications [11], referring [12], updating [13], call flow examples for VoIP [14], and PSTN [15], QoS and authorization [16], privacy [17], security [18], and many more.

In the context of VoIP, SIP is responsible for the whole user localization and call management. On behalf of a user, a User Agent (UA), typically a hard phone on the desk, or a soft phone running on some PC, sends REGISTER messages to a local registrar (server). The messages contain the ID of the user and the IP address of the UA. The registrar then updates the received locality information in yet another server called location server, which from this time on knows the address a certain user can be reached at (see Figure 1). RFC 3261

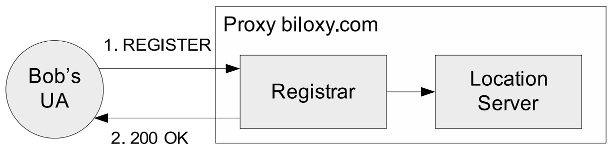

**Fig. 1.** Bob's UA registers Bob's location at the local registrar/proxy.

defines that only certain header fields are necessary for a SIP message to work properly, but of course there are many optional header fields that can be used by a VoIP phone within a SIP message as well. According to RFC 3261, a SIP REGISTER message has to contain a request line and the header fields *To*, *From*, *Call-ID* and *CSeq* (see Figure 2). Furthermore, SIP requests like INVITE messages additionally have to contain the header fields *Max-Forwards* and *Via* (see Figure 4).

```
REGISTER sip:Domain SIP/2.0
To: <sip:UserID@Domain>
From: <sip:UserID@Domain>
Call-ID: NDYzYzMwNjJhMDRjYTFj
CSeq: 1 REGISTER
```

**Fig. 2.** A typical SIP REGISTER message.

Handling calls is then the task of a so-called proxy server, which also should be installed locally at each site. Calls are initiated by sending INVITE messages. A call from user Alice registered at site atlanta.com to user Bob registered at site biloxy.com is usually done by specifying a SIP URI like bob@biloxy.com. Alice's UA then sends a SIP INVITE message to her local proxy atlanta.com, which then forwards the INVITE to Bob's proxy biloxy.com, which subsequently asks the local location server where to find Bob's UA. In case Bob's UA has been registered previously, the proxy at biloxy.com forwards the INVITE to Bob's UA, which then may continue to initiate the session. Figure 3 shows this forwarding of INVITE messages, the rest of the messages necessary to establish a call, i.e., "180 RINGING", "200 OK", and "ACK" are not shown here. Often,

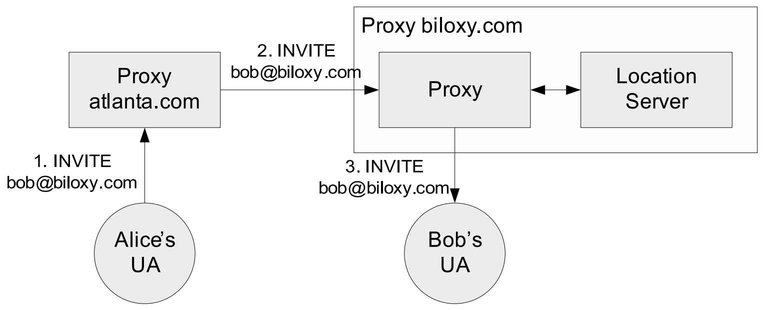

**Fig. 3.** Alice tries to setup a call to Bob.

instead of installing physically three different servers, it suffices to integrate the functionality of registrar, location server, and proxy into one server, the local proxy, which we will assume here. In particular, we assume that there is only one single SIP parser responsible for registration and call management.

```
INVITE sip:CalleeUserID@CalleeDomain SIP/2.0
To: <sip:CalleeUserID@CalleeDomain>
From: <sip:CallerUserID@CallerDomain>
Via: SIP/2.0/UDP IPAdress:Port
Call-ID: NDYzYzMwNjJhMDRjYTFj
CSeq: 1 INVITE
Max-Forwards: 70
```

**Fig. 4.** A typical SIP INVITE message.

## 4 Autonomic SIP Adaptation

Standardization of network protocols enables interconnection, however, the principle problem with open protocols lies in the degree of freedom allowed causing different protocol dialects in practice. These small differences in the messages' header information might lead to incompatibilities between implementations of different vendors and organizations, a problem that is usually gradually solved over a time period of years.

We attack the problem of transient incompatibilities by introducing a self-learning module which can be added to arbitrary proxies. The purpose of this module is twofold: first, the module should *classify* an incoming message by analyzing its header information in order to predict a rejection by the proxy and, second, the module *suggests an adaptation* of the header information which should finally force the acceptance of the message.

### 4.1 C4.5 Decision Trees

For classification, we use a C4.5 decision tree [19] capable of further identifying relevant header parameters causing rejections. Additionally, further properties of C4.5 trees seem to be desirable, like avoiding over-fitting of the tree and dealing with incomplete data. After a training phase, new messages can then be classified into messages that are likely to be rejected or accepted. The C4.5 decision tree implementation (J48) used is based on the Weka machine learning library [20]. All headers, header fields, and standard values (as defined by RFC 3261) are defined as attributes. For each attribute a *numerical value* is defined to describe a SIP message (274 attributes per message) as shown in the algorithm depicted in Figure 5. It must be noted that the result is a vector of dimension $d = 274$, vector components are 0 if the corresponding header/header field is not present, 1 if the header/header field is present and of type string (but 0 if the header format is incorrect), and any numeric value if the header/header field is present and of numeric type. For the current implementation, this information is stored in the format ARFF (Attribute Relation File Format).

```
Input:  attribute vector A
Output: attribute vector A with new values

FOREACH (Ai in A)
  Ai.value = 0
  IF (Ai.name in SIP message) THEN
    IF (SIP message field is numeric) THEN
      Ai.value = value of SIP message field
    ELSE Ai.value = 1
```

**Fig. 5.** Translation of SIP header into C4.5 attribute values.

Figure 6 shows an example tree generated by the training with SIP REGISTER messages. The tree actually represents a hierarchy of nested if-then rules each SIP message is tested against. The leaves of the tree represent the tree classification decision, in this case either ACCEPTED or REJECTED. For example, if the header field *Replaces* is in the SIP message (rule "Replaces $> 0$"), the message is classified as rejected. The rule hierarchy that was learned also shows the importance of the message's parameters for the final acceptance / rejection of the message, with important rules being at the top of the rule hierarchy. The numbers calculated for the tree leaves correspond to the number of messages which have been classified in this branch (the second number shows the number of wrong classifications in case they exist).

```
Replaces <= 0
| Allow_DO <= 0
| | Content-Language <= 0
| | | Contact_methods <= 0
| | | | To_user <= 0: ACCEPTED (112.0/5.0)
| | | | To_user > 0
| | | | | Contact_q <= 0
| | | | | | Call-ID <= 0: REJECTED (2.0)
| | | | | | Call-ID > 0: ACCEPTED (12.0/1.0)
| | | | | Contact_q > 0: REJECTED (2.0)
| | | Contact_methods > 0
| | | | Accept <= 0: REJECTED (6.0/1.0)
| | | | Accept > 0: ACCEPTED (11.0/1.0)
| | Content-Language > 0
| | | Contact_flow-id <= 0: REJECTED (6.0/1.0)
| | | Contact_flow-id > 0: ACCEPTED (2.0)
| Allow_DO > 0
| | Allow-Events_talk <= 0: REJECTED (6.0)
| | Allow-Events_talk > 0: ACCEPTED (3.0/1.0)
Replaces > 0: REJECTED (11.0)
```

**Fig. 6.** Example C4.5 tree after training with SIP REGISTER messages.

## 4.2 Babel-SIP

Babel-SIP is an automatic protocol adapter and is placed between the proxy socket that accepts the incoming messages, and the registrar's or proxy's SIP parser (see Figures 7 and 8). Babel-SIP maintains a C4.5 decision tree, and observes which messages are accepted by the proxy, and which are not. This information is fed into the decision tree, the tree thus learns which headers are likely to cause trouble for this particular release of the proxy software.

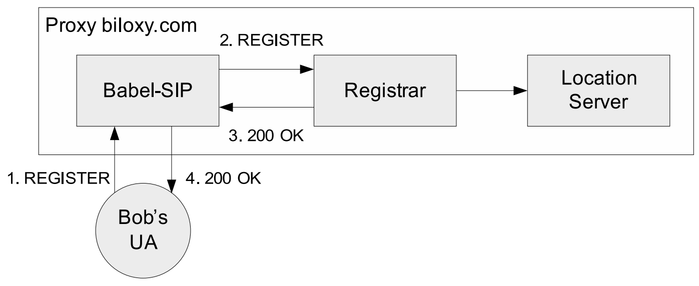

**Fig. 7.** Babel-SIP adapts incoming REGISTER messages and passes them on to a registrar.

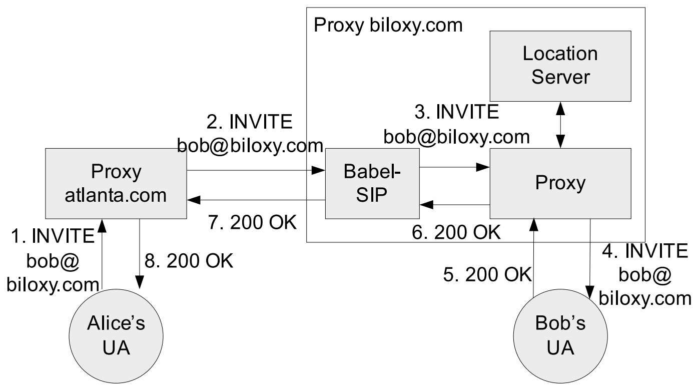

**Fig. 8.** Babel-SIP adapts incoming INVITE messages and passes them on to the proxy.

Once the tree has been trained, by using the same decision tree, incoming messages are then automatically classified as either probably accepted or probably rejected. Of course, at this stage, Babel-SIP does not know this for sure. However, once a message is classified to be probably rejected by the proxy parser, Babel-SIP tries to adapt SIP messages in such a way that the result turns into a probably accept.

Babel-SIP stores messages that have been accepted previously by the proxy in a local database $M$. Once a SIP message $m_1$ has been classified as probably rejected, Babel-SIP searches through its database for the message $m_c$ being closest to $m_1$. For estimating the distance between two SIP messages $m_1$ and $m_2$, we use the standard Euclidean distance metric $d(m_1, m_2)$ provided by Weka (on normalized versions of the vectors). The sought for message $m_c$ is thus given by

$$m_c = \arg \min_{m_i \in M \wedge m_i \neq m_1} d(m_1, m_i).$$

Babel-SIP then identifies those headers of $m_1$ which are classified as being problematic. This information is again derived from the decision tree. If the same header/header field is found in $m_c$ then the according header/header field/values of $m_c$ are copied into $m_1$, thus replacing the previous information. If the header/header field is not found in $m_c$, it is erased from $m_1$. Furthermore, Babel-SIP identifies those headers and header fields of $m_c$ which are not used in $m_1$, and inserts them into $m_1$. The result is a new message version $\hat{m}_1$, which is then forwarded to the proxy.

At this point it must be noted that it is self-evident that such an approach must be done with care. In a real production system it is mandatory that the appropriate semantics of the different headers are also taken into account which we have not addressed so far. Rather, the aim of this work is to evaluate whether our approach based on observing and learning is able to achieve an improved rate of message acceptance or not. In our follow-up work we will thus focus on creating appropriate rules for semantics-aware header translation.

## 5 Experiments and Results

In our lab we installed several popular hard and soft phones and ran experiments using a commercial proxy server created by a major Austrian telecom equipment provider. At the time of our research, this commercial proxy was guaranteed to work with only two different types of VoIP hard phones and only one type of VoIP soft phone. For all other phones the company did not guarantee that the phones would work with their SIP proxy, although mostly they did.

Our research in this work primarily focuses on the REGISTER and INVITE messages. In the initial phase we aimed at finding out how compatible our proxy is with respect to different versions of REGISTER messages. In the second phase we concentrated on probably the most important SIP message, the INVITE message. In our experiments we found a multitude of different SIP headers used by different phones, so our first step was to find out what kind of REGISTER and INVITE messages the different types of VoIP phones send. We therefore monitored REGISTER and INVITE messages from nine popular VoIP hard phones and five popular VoIP soft phones (see Table 1).

After these first experiments we found out that the SIP messages can indeed be very different. For example: one hard phone $A$ uses less than 10 header fields in its SIP REGISTER message, another hard phone $B$ uses 15 header fields in its SIP REGISTER message, and both phones use different header fields as well.

### 5.1 Initial SIP Messages

In order to obtain a substantial amount of possible REGISTER and INVITE messages for testing our proxy, we decided to artificially create different SIP messages. The newly generated messages were random combinations of the observed SIP header fields, header field values, and header field parameters from the investigated real phones, as well as others taken from RFC 3261.

For both REGISTER and INVITE messages we generated a set of the 53 most often used SIP header fields found in the observed SIP messages. For these 53 header fields we defined 145 header field values and header field parameters.

**Table 1.** Hard and soft phones analyzed for our experiments.

| | |
|---|---|
| **Hard Phones** | Snom 300 |
| | Polycom SoundPoint IP 330 |
| | Linksys SPA IP Phone SPA 941 |
| | DLink VoIP IP-Phone DPH-120S |
| | Thomson ST2030 |
| | Allnet 7950 (Sipgate) |
| | Grandstream GXP2000 Enterprise IP Phone |
| | Elmeg IP 290 |
| | Siemens |
| **Soft Phones** | X-Lite |
| | PortSIP |
| | BOL |
| | Express Talk |
| | 3CX |

In the next step we generated different SIP REGISTER and INVITE messages using these 145 header field values. The basic idea was to generate a large amount of different messages with many different parameter combinations. We therefore defined a probability for each of the 145 different header fields, the probability values were computed from the previously observed SIP messages sniffed from the real phones. Furthermore we developed a Java program that continuously sends either REGISTER or INVITE messages to the proxy, the messages being created randomly by choosing a subset of the 145 header field values according to their probabilities.

In our experiments we generated 344 different SIP REGISTER messages and 122 different SIP INVITE messages. For each SIP REGISTER message we determined whether the register process was successful, i.e., whether the client received a "200 OK" reply, or not. If it was successful we marked the message as *accepted*, otherwise we marked it as *rejected*. Out of the 344 REGISTER messages, 78 (or approximately 22.67%) were marked as rejected, as it turned out, mostly because of incomplete header information. For each SIP INVITE message we determined whether the call setup was successful and the INVITE message was successfully sent to the called party, i.e., whether the client received a "180 Ringing" reply, or not. Again, if it was successful we marked the message as accepted, otherwise we marked it as rejected. Out of the 122 INVITE messages, 69 (or approximately 56.56%) were marked as being rejected.

### 5.2 Rejected Messages

We ran several experiments in our lab to evaluate the effectiveness of Babel-SIP, which is measured by the improvement of acceptance of previously not accepted SIP messages. These experiments were carried out separately for REGISTER and INVITE messages.

**REGISTER Messages** Initially, a C4.5 decision tree was built from the *training data set* composed of 50 messages (of which 22% are known to be rejected) randomly selected from the artificial messages generated. This initial tree classifies 90% of the training data set correctly.

Then we ran a set of experiments, consisting of 15 replicated experimental runs. In each run we sent a total of 4400 messages to Babel-SIP, chosen at random from the set of 294 test messages. Here, 22.79% of the test data messages were known to be rejected by the proxy. The partitioning of messages into a training and test set is shown in Table 2.

**Table 2.** Initial training and test data sets.

| REGISTER messages | | | |
|---|---|---|---|
| Training data set | Accepted messages | 39 | 78% |
| | Rejected messages | 11 | 22% |
| | Total number | 50 | |
| Test data set | Accepted messages | 227 | 77.21% |
| | Rejected messages | 67 | 22.79% |
| | Total number | 294 | |

| INVITE messages | | | |
|---|---|---|---|
| Training data set | Accepted messages | 9 | 45% |
| | Rejected messages | 11 | 55% |
| | Total number | 20 | |
| Test data set | Accepted messages | 44 | 43.14% |
| | Rejected messages | 58 | 56.86% |
| | Total number | 102 | |

Since training is very resource consuming, we further decided that the decision tree is not trained after each single message. Rather, the results of a batch of 20 consecutive REGISTER messages were used to train the decision tree, which as a result gradually adapted to the acceptance behavior of the proxy.

For each message we recorded whether it was accepted or not. The 15 experiments thus resulted in 15 time series of 4400 binary observations (yes or no). For each experiment $l, 1 \leq l \leq 15$ we then calculated rejection rates over overlapping bins of size 100 messages. The first bin $B_1^l = \{m_i^l \,|\, 1 \leq i \leq 100\}$ includes messages 1 to 100, the rejected messages of this bin are given by $\hat{B}_1^l = \{m_i^l \in B_1^l \,|\, m_i^l \text{ was rejected}\}$. $B_2^l$ and $\hat{B}_2^l$ are then computed over messages 21 to 120 from experiment $l$. In general, for $1 \leq k \leq 216$ we define

$$B_k^l = \{m_i^l \,|\, 20 \times (k-1) + 1 \leq i \leq 20 \times (k-1) + 100\}$$

and

$$\hat{B}_k^l = \{m_i^l \in B_k^l \,|\, m_i^l \text{ was rejected}\}.$$

Thus, an estimator $\hat{R}^l(i)$ for the rejection rate (in %) around message $m_i^l, 1 \leq i \leq 4400$ is given by

$$\hat{R}^l(i) = 100 \times |\hat{B}^l_{\lceil i/20 \rceil}|/|B^l_{\lceil i/20 \rceil}| = |\hat{B}^l_{\lceil i/20 \rceil}|.$$

Figure 9 shows a smoothened curve of the estimated rejection rate. By using

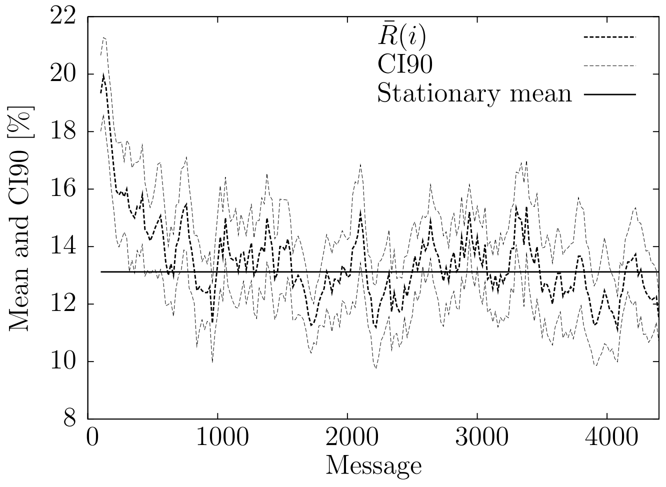

**Fig. 9.** Mean estimated rejection rate $\bar{R}(i)$ and 90% confidence interval (CI90) for message $i$. $\bar{R}(i)$ is calculated over all 15 REGISTER experiments and each bin. Stationary mean is the overall mean for messages $m_{600}^l$ to $m_{4400}^l$.

$$\bar{B}_k = \left( \sum_{l=1}^{15} |\hat{B}_k^l| \right) /15, \; 1 \leq k \leq 216, \tag{1}$$

we define a mean estimator by

$$\bar{R}(i) = \bar{B}_{\lceil i/20 \rceil},$$

i.e., the mean is calculated for each bin over all 15 experimental runs. In Figure 9 it can be seen that the time series shows a transient phase at the start, in which the rejection rate decreases. In this phase, Babel-SIP gradually learns and increases its effectiveness. The series then enters a stationary phase, in which no more gain is achieved, i.e., Babel-SIP has seen all possible messages. We have additionally computed the mean rejection rate $R^s \approx 13.12\%$ for messages in this stationary phase, taking into account only messages $m^l_{600}$ to $m^l_{4400}$ (Stationary mean). This rejection rate is the main result of Babel-SIP: when sending our test data to the commercial proxy, the application of Babel-SIP is able to decrease the rejection rate from 22.79% to 13.12%, i.e., the rejection rate on average is decreased by over 42%.

Table 3 shows aggregated results over all 15 experiments in more detail. Modified denotes the percentage of modified messages in relation to all messages. Successfully modified denotes those messages that were turned from a rejected into an accepted message by Babel-SIP. This statistic is given one time in relation to all messages, and one time in relation only to those messages that have been classified as rejected. This number is the number of rejected messages minus the number of false negatives, plus the number of false positives. The false positives are those that were classified as rejected, although they were not. False negatives are those messages that were classified as accepted although they were not. For these statistics the table shows the mean over all 15 runs, as well as the standard deviation between the individual runs and this mean. The standard deviations are very small, and thus on the aggregate level, all 15 experiments show almost equal results.

**Table 3.** Aggregated results of Babel-SIP experiments.

| REGISTER messages | Mean [%] | Std.dev. [%] |
|---|---|---|
| Modified | 18.37 | $5.8310^{-4}$ |
| Successfully modified (from all) | 9.33 | 0.003 |
| Successfully modified (of those classified as rejected) | 48.02 | 0.108 |
| False positive | 6.13 | $2.04110^{-4}$ |
| False negative | 10.53 | $4.23310^{-4}$ |

| INVITE messages | Mean [%] | Std.dev. [%] |
|---|---|---|
| Modified | 38.24 | 0 |
| Successfully modified (from all) | 22.39 | 1.210 |
| Successfully modified (of those classified as rejected) | 58.55 | 3.165 |
| False positive | 6.86 | 0 |
| False negative | 23.53 | 0 |

**INVITE Messages** The experiments to validate our approach on INVITE messages were performed similarly to the REGISTER experiments. The initial C4.5 tree built from the training data set sorts all 20 training messages correctly into accepted and rejected ones. In each of the 15 experimental runs, 1000 INVITE messages were randomly selected from the test data set and sent to Babel-SIP. 56.86% of the test messages (see Table 2) are known not to be accepted by the SIP proxy. The decision tree used to predict whether an incoming message will be accepted was again updated after every 20 messages that were sent to the proxy. The rejection rates $\hat{R}^l(i)$ for two experiments are shown in Figure 10.

For the 15 INVITE experiments we again computed a mean estimator $\bar{R}(i)$ and the 90% confidence interval, as well as an estimator for the stationary rejection rate (see Figure 11). For the latter we used messages $m^l_{560}$ to $m^l_{1000}$. This stationary rejection rate turned out to be 30.79%. When compared to the initial rejection rate of 56.86%, this is an improvement of 45.85%.

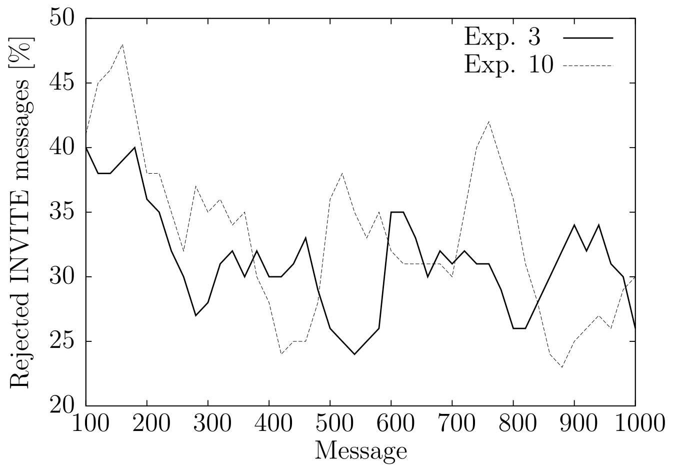

**Fig. 10.** Estimated rejection rates $\hat{R}^l(i)$ for two INVITE experiments.

Details on the modified and incorrectly classified messages in all 15 experimental runs are given in Table 3. The number of messages wrongly classified as rejected (false positives) is as high as in the REGISTER experiments, whereas the number of those wrongly classified as accepted (false negatives) has strongly increased. Thus, it can be said that the decision tree classifying the INVITE headers is not as accurate in distinguishing the rejected headers from the accepted headers as the tree resulting from the REGISTER experiments.

Another interesting observation is the observed zero standard deviation in the percentage of messages classified as rejected and, subsequently, modified over all 15 experimental runs. As a result, it seems that the trees classified all messages equally although the messages were selected in a random sequence for each run

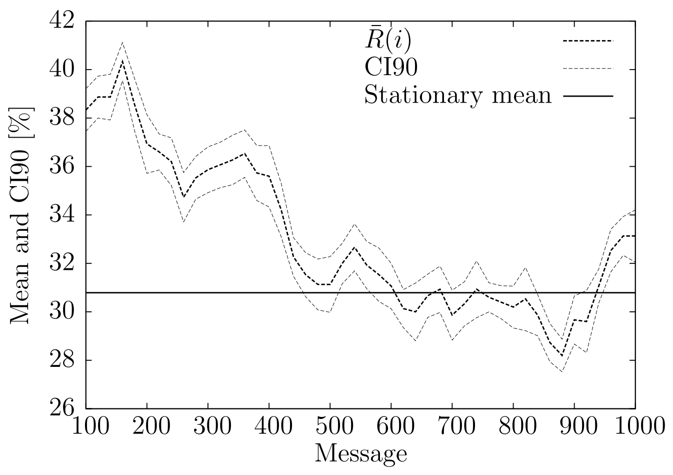

**Fig. 11.** Mean estimated rejection rate $\bar{R}(i)$ and 90% confidence interval (CI90) for message $i$. $\bar{R}(i)$ is calculated over all 15 INVITE experiments and each bin. Stationary mean is the overall mean for messages $m^l_{560}$ to $m^l_{1000}$.

and each tree should thus learn some faulty parameters earlier and some later than trees in other runs. However, in the REGISTER experiments the observed variances for the same statistic were also quite small, and it must be noted that in the REGISTER experiments both a much larger test message pool (294 vs. 102) and much longer experimental sequences (4400 vs. 1000) were used.

## 5.3 Retries

Since the basic hypothesis of Babel-SIP is that even phones which initially fail to contact the SIP proxy, after retrying a number of times, eventually may succeed because of Babel-SIP learning and transformation, we have analyzed the number of attempts that are necessary for succeeding. For this, we ran separated REGISTER and INVITE experiments with our training and test data sets (see Table 2). Each experiment was driven by a parameter $r$, stating the maximum number of times a phone would try to register itself or to call another phone. Babel-SIP was again trained before the experiments were started by sending the test data. In the case that the phone had not succeeded within these tries, it was counted as "never". For each of the 67 REGISTER respectively 58 INVITE messages of initially rejected test messages we recorded the number of messages necessary for registering and inviting.

Table 4 shows the number of necessary tries up to that value of $r$ from which onwards no improvement of the number of accepted messages was achieved. The results reveal that for those phones that would need more than one try, most of them would succeed after at most four tries. Mapping this onto a realistic scenario, this means that a phone owner would have to attempt to register his phone or to initiate a call only a few times, if he would be willing to wait between the tries for some time. In addition, it was noticed that the number of accepted messages among the previously rejected ones did not increase if the phones repeatedly attempt to register more than 9 times or to start a call more than 4 times.

**Table 4.** Number of necessary attempts for experiment $r$.

| # REGISTERs | **1** | **2** | **3** | **4** | **$\geq$ 5** | **never** |
|---|---|---|---|---|---|---|
| $r = 1$ | 10 | | | | | 57 |
| $r = 2$ | 14 | 8 | | | | 45 |
| $r = 3$ | 14 | 12 | 3 | | | 38 |
| $r = 4$ | 21 | 5 | 3 | 1 | | 37 |
| $r = 5$ | 17 | 7 | 5 | 2 | 0 | 36 |
| $r = 6$ | 16 | 13 | 2 | 0 | 0 | 36 |
| $r = 7$ | 19 | 9 | 1 | 2 | 0 | 36 |
| $r = 8$ | 16 | 5 | 7 | 2 | 1 | 36 |
| $r = 9$ | 18 | 7 | 5 | 2 | 0 | 35 |
| # INVITEs | **1** | **2** | **3** | **$\geq$ 4** | | **never** |
| $r = 1$ | 22 | | | | | 36 |
| $r = 2$ | 25 | 3 | | | | 30 |
| $r = 3$ | 21 | 5 | 2 | | | 30 |
| $r = 4$ | 25 | 1 | 5 | 1 | | 26 |

### 5.4 Rejection of Previously Accepted Messages

An important evaluation criterion is given by the question, whether Babel-SIP actually may wrongly classify a message $m$ as probably rejected, while in reality the message would be accepted, and would change it in a way that the new version $\hat{m}$ would be actually rejected. Such a behavior is regarded as being unacceptable, since proxy providers provide a list of evaluated UAs to their customers. A translation component with such an erroneous behavior would nullify any such guarantee.

As a consequence, we ran additional experiments and added the SIP messages from 9 hard phones and 5 soft phones to the pool of our test data. The phone messages were known to be accepted by the proxy. There was not a single instant that any of these real phone messages were altered and subsequently rejected by the proxy. The same is true for all REGISTER and INVITE test messages from the accepted pool. At least our experiments with the given proxy showed that Babel-SIP indeed shows a stable behavior, and tends to change only messages which are problematic, but does not alter already accepted messages in such a way that they subsequently would be rejected.

## 6 Qualitative Analysis of Decision Trees

Since the C4.5 decision tree contains a summary of the SIP parser behavior, it also can be used as a hint for the proxy programmers to find bugs in their implementation. This will be demonstrated in the following sections.

### 6.1 REGISTER Messages

As mentioned above, after the initial training with 50 REGISTER messages the C4.5 tree is able to classify 90% of the messages correctly.

After 70 REGISTER messages, i.e., the initial training (50 messages) plus one more training (20 more messages), the resulting tree is quite small, containing only five rules which decide whether an incoming message is classified as accepted or rejected (see Figure 12). Looking at this tree, a programmer can already see important hints explaining failures of his SIP parser. The accuracy of this tree is already 91.43%.

```
Event_message-summary <= 0
| Error-Info <= 0
| | Contact_transport <= 0: ACCEPTED (52.0/4.0)
| | Contact_transport > 0
| | | Via_rport <= 0: REJECTED (2.0)
| | | Via_rport > 0: ACCEPTED (8.0/1.0)
| Error-Info > 0
| | Allow-Events_presence <= 0: REJECTED (2.0)
| | Allow-Events_presence > 0: ACCEPTED (2.0)
Event_message-summary > 0: REJECTED (4.0/1.0)
```

**Fig. 12.** Example C4.5 tree after training with 70 REGISTER messages.

Looking at the C4.5 decision tree derived at the end of the experiments (see Figure 13) the tree gets quite large, thus using this tree for analysis imposes more work to the programmer. On the other hand, its classification accuracy is increased to 99.32%. Thus, it can be assumed that this tree indeed sufficiently explains registration failures without knowing anything about the implementation details.

Because the final tree is quite large, for readability reasons, Figure 13 shows just the parts containing the most important rules as outcome of our experiments. As mentioned in Section 5, the developed Java client creates artificial messages, and each of the 145 header fields/values is included only according to some preset probability. Therefore it is possible that important header fields or header field values are not used, and some messages do not contain all mandatory header fields as demanded by RFC 3261. We thus investigate whether the decision tree can be used to understand why the parser rejected many of the rejected messages.

In the decision tree depicted in Figure 13, the first rule indicates that a SIP REGISTER message has to contain the header field *CSeq* (which actually is mandatory as described in RFC 3261). If an incoming message does not contain a *CSeq* header field it is very likely to be rejected, otherwise the next rule has to be checked. Since the tree represents a hierarchy of rules, the tree can be traversed from top to bottom, or vice versa. When checking the C4.5 tree from top to bottom, the next rule indicates that a SIP message has to contain the header field *Call-ID*, again according to RFC 3261. We conclude that checking the decision tree from top to bottom results in precise hints about the critical header fields. Of course as the tree gets larger there are more and more branches and rules to be checked, therefore sometimes after the first quick check from top to bottom it makes sense to check the tree from bottom to top. In most cases it comes down to a final rule that indicates whether a message is likely to be accepted or rejected. Figure 13 shows that the *Via* header field parameter *rport* is such a final decision rule. This means that at a specific condition set by previous rules, the *rport* parameter of the *Via* header is mandatory for a SIP REGISTER message to work properly. Checking the decision tree from top to

```
CSeq <= 0: REJECTED (104.0)
CSeq > 0
| Call-ID <= 0: REJECTED (101.0)
| Call-ID > 0
| | Replaces <= 0
| | | [...]
| | | To_IP <= 0
| | | | [...]
| | | To_IP > 0
| | | | [...]
| | | | | Via_IP <= 0
| | | | | | [...]
| | | | | Via_IP > 0
| | | | | | [...]
| | | | | | From_IP <= 0: REJECTED (2.0)
| | | | | | From_IP > 0: ACCEPTED (38.0)
| | | | | | [...]
| | | | | | Via_rport <= 0: REJECTED (11.0)
| | | | | | Via_rport > 0: ACCEPTED (30.0)
| | | | | | [...]
| | Replaces > 0: REJECTED (38.0)
```

**Fig. 13.** Part of a C4.5 tree (SIP REGISTER messages).

bottom or vice versa results in a set of rules enabling the programmer either to find bugs in a proxy's implementation or to find out why incoming SIP messages were rejected. To show the simplicity of the C4.5 decision tree we checked all 78 different faulty SIP REGISTER messages and tried to find out why they were rejected by the SIP proxy. With 51 messages (or 65.38% of all faulty messages) and with the help of the decision tree it is extremely easy to find out why the message was rejected, even for people without detailed knowledge of the RFC 3261.

## 6.2 INVITE Messages

Since in our experiments the INVITE messages did cause errors different from those of the REGISTER messages, the decision trees stemming from the INVITE experiments were quite different from those stemming from the REGISTER experiments. Figure 14 shows that incoming INVITE messages that do not contain

```
To_IP <= 0: REJ. (72.0)
To_IP > 0
| Replaces <= 0
| | [...]
| | From_IP <= 0
| | | Allow-Events_conference <= 0: REJ. (31.0)
| | | Allow-Events_conference > 0: ACC. (10.0)
| | From_IP > 0
| | | [...]
| | | Content-Type_text/html <= 0: ACC. (18.0)
| | | Content-Type_text/html > 0: REJ. (2.0)
| | | [...]
| | | Privacy_header <= 0: ACC. (58.0/2.0)
| | | Privacy_header > 0
| | | [...]
| | | Request-Line_transport <= 0: REJ. (25.0)
| | | Request-Line_transport > 0: ACC. (9.0)
| | | [...]
| Replaces > 0: REJ. (48.0)
```

**Fig. 14.** Part of a C4.5 tree (SIP INVITE messages).

the *To_IP* parameter, i.e., the *domain* in the *To* header, will probably be rejected by the SIP proxy. The *From_IP*, meaning the *domain* in the *From* header, is also an important factor if the message will be accepted or rejected.

Similar to the REGISTER case, the decision tree derived at the end of the INVITE experiments shows lots of final decision rules. Figure 14 shows that the tested SIP proxy has problems with INVITE messages either containing the "text/html" value in the *Content-Type* header, containing the *Privacy* header, or containing the "transport=udp" value in the *Request-Line*.

Figure 13 as well as Figure 14 show that both REGISTER and INVITE messages are likely to be rejected by the tested SIP proxy if the message contains the *Replaces* header. In our follow-up work we intend to test other open-source SIP proxies with the same set of SIP messages to see whether there are notable differences in this behavior.

## 7 Conclusion

In this paper we introduce Babel-SIP, an automatic, self-learning SIP-message translator. Its main use is in the transient phase between creating a new SIP-stack implementation or a whole new proxy, and the final release of a 100% reliable proxy version. Generally such situations occur immediately after the development of a new communication protocol which is quickly adopted and implemented by numerous vendors. Due to the immaturity of software implementations, during this phase, devices often fail to communicate, although in theory they implement the same protocol.

The task of Babel-SIP is to act as a mediator for a specific release of a SIP proxy. Babel-SIP analyzes SIP messages sent to its proxy (currently REGISTER and INVITE messages), and learns which SIP messages were accepted by this proxy, and which were not. Over time, Babel-SIP is able to accurately guess whether an incoming messages is likely to be accepted by its proxy. If not, Babel-SIP changes the message in such a way that the probability for acceptance is increased. By carrying out numerous experiments, we have demonstrated that with our approach the number of rejected messages for both SIP message types is almost halved, and that Babel-SIP gains knowledge over time and improves its effectiveness.

Additionally we have shown that the resulting decision trees indeed provide good insight into the faulty behavior of either the SIP parser or the SIP clients and phones themselves. As a consequence, the decision trees can be used by SIP programmers to remove implementation bugs.

In the near future we will focus on creating semantic rules for changing header information and test Babel-SIP with different SIP proxies. Since we regard our approach to be generic, we will also investigate the possible application of our Babel approach to other popular application protocols such as RTSP or HTTP.

## Acknowledgment

The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsfoerderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

## References

1. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: SIP: Session Initiation Protocol. RFC 3261 (June 2002)

2. Wilking, D., Röfer, T.: Realtime Object Recognition Using Decision Tree Learning. In: RoboCup 2004: Robot World Cup VII, Springer (2005) 556–563
3. Heisig, S., Moyle, S.: Using Model Trees to Characterize Computer Resource Usage. In: 1st ACM SIGSOFT Workshop on Self-Managed Systems. (2004) 80–84
4. Abbes, T., Bouhoula, A., Rusinowitch, M.: Protocol Analysis in Intrusion Detection Using Decision Trees. In: International Conference on Information Technology: Coding and Computing (ITCC'04). (2004) 404–408
5. Kang, H., Zhang, Z., Ranjan, S., Nucci, A.: SIP-based VoIP Traffic Behavior Profiling and Its Applications. In: MineNet'07. (2007) 39–44
6. Aichernig, B., Peischl, B., Weiglhofer, M., Wotawa, F.: Protocol Conformance Testing a SIP Registrar: an Industrial Application of Formal Methods. In: 5th IEEE Int. Conference on Software Engineering and Formal Methods. (2007) 215–224
7. Abdelnur, H., State, R., Festor, O.: KiF: A Stateful SIP Fuzzer. In: 1st Int. Conference on Principles, Systems and Applications of IP Telecommunications, iptcomm.org (2007)
8. Acharya, A., Wand, X., Wrigth, C., Banerjee, N., Sengupta, B.: Real-time Monitoring of SIP Infrastructure Using Message Classification. In: MineNet'07. (2007) 45–50
9. Rosenberg, J., Schulzrinne, H.: Reliability of provisional responses in session initiation protocol (sip). RFC 3262 (June 2002)
10. Rosenberg, J., Schulzrinne, H.: Session initiation protocol (sip): Locating sip servers. RFC 3262 (June 2002)
11. Roach, A.B.: Session initiation protocol (sip)-specific event notification. RFC 3265 (June 2002)
12. Sparks, R.: The session initiation protocol (sip) refer method. RFC 3515 (April 2003)
13. Rosenberg, J.: The session initiation protocol (sip) update method. RFC 3311 (September 2002)
14. Johnston, A., Donovan, S., Sparks, R., Cunningham, C., Summers, K.: Session initiation protocol (sip) basic call flow examples. RFC 3665 (December 2003)
15. Johnston, A., Donovan, S., Sparks, R., Cunningham, C., Summers, K.: Session initiation protocol (sip) public switched telephone network (pstn) call flows. RFC 3666 (Decmember 2003)
16. Marshall, W.: Private session initiation protocol (sip) extensions for media authorization. RFC 3313 (January 2003)
17. Peterson, J.: A privacy mechanism for the session initiation protocol (sip). RFC 3323 (November 2002)
18. Arkko, J., Torvinen, V., Camarillo, G., Niemi, A., Haukka, T.: Security Mechanism Agreement for the Session Initiation Protocol (SIP). RFC 3329 (January 2003)
19. Mitchell, T.: Machine Learning. Mc-Graw-Hill (1997)
20. Witten, I., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann (2005)