Babel-SIP: Self-learning SIP Message Adaptation for Increasing SIP-Compatibility

Helmut Hlavacs, Karin Anna Hummel, Andrea Hess, and Michael Nussbaumer
University of Vienna
Department of Distributed and Multimedia Systems
Vienna, Austria
Email: {helmut.hlavacs | karin.hummel | andrea.hess | michael.nussbaumer} @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, standard compliant messages 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 can be put in front of the actual SIP parser of a SIP proxy. By training a C4.5 decision tree, it gradually learns, which SIP messages are accepted by the parser, and which are not. The same tree can then be used for classifying incoming SIP messages. Those classified as "not accepted" can then be pro-actively changed into the most similar message that is known to be accepted from the past. By running experiments using a commercial SIP proxy, we demonstrate that Babel-SIP can drastically increase the message acceptance rate.**

I. Introduction

One of the success factors of the Internet and of many of its applications is the openness of its protocols. Thousands of protocols exist 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 comprising 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. As a consequence, incompatibilities may occur between communicating peers.

Consider SIP [8] for instance, which will be introduced in more detail in Section III. 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. 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 in their offices. 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.

Of course, in the long run, products get more and more robust, and compatibility issues are gradually removed. However, in the transient phase of 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 only. Rather, it is thinkable to construct other versions of Babel-SIP for newly proposed protocols in order to improve transient situations for newly deployed products.

II. Related Work

Since protocols are either proprietary or standardized, using autonomous, self-adapting parsers based on machine learning techniques are not as widespread as in other domains, such as robotics, natural language classification and learning, node and network utilization and prediction of future utilization (as needed in, e.g., Grid environments), 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) [9] or characterization of computer resource usage [5].

In [1] 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.

In the application area of VoIP and SIP, authors both investigate traffic behavior and failures in particular software implementations. In [6] 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 [4] 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 [2] a fuzzer was added to a SIP proxy installation similarly to our system architecture. But, instead of analyzing incoming traffic, the module was used to systematically test a SIP proxy with different (faulty) messages both in terms of syntax and in terms of protocol behavior. The work closest to our approach is presented in [3]. 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.

III. 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 [8] but several other RFCs define different additional aspects of SIP, as found for instance in the RFCs 3262, 3263, 3265, 3515, 3311, 3665 and others.

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 this site (see Figure 1). RFC 3261 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. Usually a SIP REGISTER message has to contain a request line and the header fields To, From, Via, Call-ID, CSeq, Max-Forwards and Contact (see Table I). 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 (not treated in this paper). Often, 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. Especially, we assume that there is only one single SIP parser responsible for registering and call management.

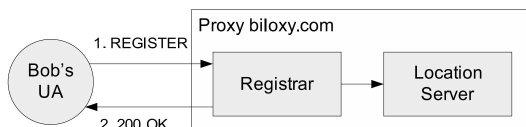

Fig. 1. Bob's UA registers Bob's location at the local registrar/proxy.

```
REGISTER sip:Domain SIP/2.0
To: <sip:UserID@Domain>
From: <sip:UserID@Domain>
Via: SIP/2.0/UDP IPAdress:Port
Call-ID: NDYzYzMwNjJhMDRjYTFj
CSeq: 1 REGISTER
Contact: <sip:UserID@IPAdress:Port>
Max-Forwards: 70
```

TABLE I
A typical SIP REGISTER message.

IV. Autonomic SIP Adaptation

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 from the proxy and, second, the module *suggests an adaptation* of the header information which should finally force the acceptance of the message.

A. C4.5 Decision Trees

For classification, we use a C4.5 decision tree [7] 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 be then 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 [10]. All headers, header fields, and standard values (as defined by SIP 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 2. For the current implementation, this information is stored in the format ARFF (Attribute Relation File Format).

Figure 3 shows an example tree generated by the training with SIP REGISTER messages. The hierarchy that was learned shows the importance of the message's parameters for the final acceptance / rejection of the message. For example, if the

```
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. 2. Translation of SIP header into C4.5 attribute values.

header field *Replaces* is in the SIP message, the message is rejected. 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 if 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. 3. Example C4.5 tree after training with SIP REGISTER messages.

B. Babel-SIP

Babel-SIP is an automatic protocol adaptor and is placed between the proxy socket that accepts the incoming messages, and the registrar's or proxy's SIP parser (see Figure 4). Babel-SIP maintains a C4.5 decision tree, and observes which

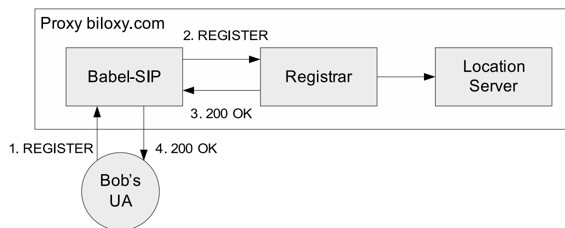

Fig. 4. Babel-SIP adapts incoming REGISTER messages and passes them on to a registrar.

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.

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 the REGISTER message in such a way that the result turns into a probably accept.

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. Additionally, Babel-SIP stores messages that have been accepted previously by the proxy in a local database M. Once a REGISTER 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, i.e.

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

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 version $\hat{m}_1$, which is then forwarded to the registrar.

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 is able to achieve an improved rate of message acceptance or not. In our follow-up work we thus focus on creating appropriate rules for header translation.

V. 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. This commercial proxy we used is 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 does not guarantee that the phone will work with their SIP proxy, although mostly they do.

Our research in this work primarily focuses on the REGISTER message. In the initial phase we aimed at finding out how compatible our proxy is with respect to different versions of REGISTER messages. 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 messages the different types of VoIP phones send. We therefore monitored REGISTER messages from the nine most popular VoIP hard phones and five most popular VoIP soft phones. Within these first experiments we saw that the SIP messages can indeed be very different. For example: one hard phone A uses less than 10 header fields in it's SIP REGISTER message, another

hard phone B uses 15 header fields in it's SIP REGISTER message, and both phones use different header fields as well.

A. Initial SIP Messages

In order to obtain a substantial amount of possible REGISTER 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.

We generated a set of the 53 most often used SIP header fields found in the REGISTER messages. Within these 53 header fields we defined 145 header field values and header field parameters. In the next step we had to generate different SIP REGISTER 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 REGISTER messages sniffed from the real phones. Furthermore we developed a Java program that continuously sends REGISTER messages to our proxy, the messages being created randomly by choosing a subset of the 145 header field values according to their probabilities.

During our experiments we generated 344 different SIP REGISTER messages. For each SIP message we determined whether the register process was successful, i.e., if the Java client received a "200 OK" reply, or not. If it was successful we marked the message with *accepted*, otherwise we marked it *rejected*. Out of the 344 messages, 78 (or approximately 22.67%) were marked as being *rejected*, as it turned out, mostly because of incomplete header information.

B. 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 REGISTER messages.

Initially, a 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 not to be accepted by the proxy. The partitioning of messages into a training and test set is shown in Table II.

Since training is very resource consuming, we further assumed 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.

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	

TABLE II
INITIAL TRAINING AND TEST DATA SETS.

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

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

Figure 5 shows the estimated rejection rate for two experiments. Though the estimator shows a quite high variance, it can be seen that there is a decrease at the start. Figure 6 shows a smoother version of the results. By using

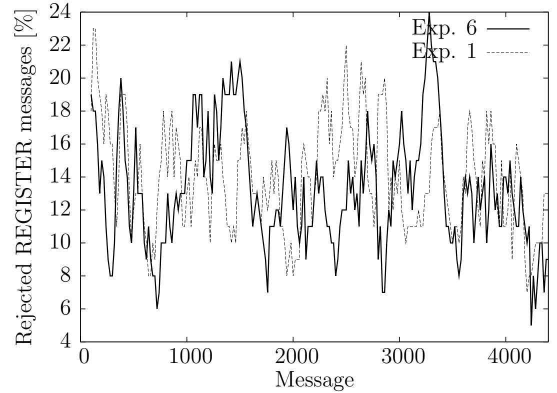

Fig. 5. Estimated rejection rates $\hat{R}^l(i)$ for two experiments.

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

we define a mean estimator by

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

i.e., the mean is calculated for each batch over all 15 experimental runs. In Figure 6 it can be seen that the time series

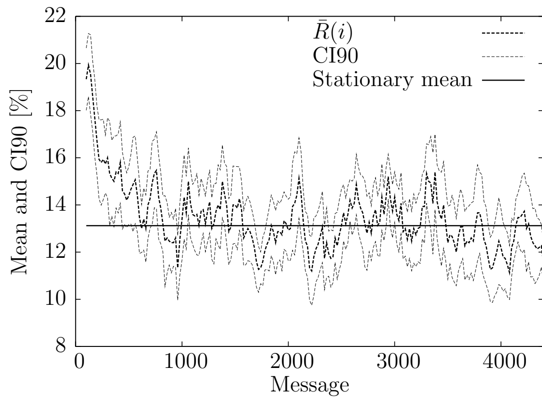

Fig. 6. Mean estimated rejection rate $\bar{R}(i)$ and 90% confidence interval (CI90) for message i. $\bar{R}(i)$ is calculated over all 15 experiments and each bin. Stationary mean is the overall mean for messages m^l_{600} to m^l_{4400}.

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 III 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 negative 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.

	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}

TABLE III
AGGREGATED RESULTS OF BABEL-SIP EXPERIMENTS.

C. Rejection of Previously Accepted Messages

An equally important metric 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 proven UAs to their customers, and 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 14 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 the 227 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 affect already accepted messages.

D. Re-REGISTERs

Since the basic hypothesis of Babel-SIP is that even phones which initially fail to register themselves, after retrying a number of times, eventually may succeed because of Babel-SIP learning and transformation. We have analyzed the number of REGISTERs that are necessary for succeeding. For this we ran further experiments, again first training Babel-SIP with our training data, then starting the experiments by sending the test data. Each experiment was driven by a parameter r, stating the maximum number of times a phone would try to register itself. If the phone does not succeed within these tries, it gives up and is counted as “never”. For each 67 messages from the initially rejected test messages (see Table II) we then recorded the number of REGISTER messages necessary for registering itself.

Table IV shows the number of necessary REGISTERs up to a maximum number of r REGISTERs. The table shows that for those phones that would need more than one REGISTER message, 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 try to register his phone only a few times, if he would be willing to wait between the tries for some time.

E. Qualitative Analysis

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.

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

After 70 messages (initial training plus one more training) the resulting tree is quite small, containing only five rules

# REGISTERs	**1**	**2**	**3**	**4**	**5**	**6**	≥ 7	**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	0		36
$r = 7$	19	9	1	2	0	0	0	36
$r = 8$	16	5	7	2	0	0	1	36
$r = 9$	18	7	5	2	0	0	0	35
$r = 10$	15	13	2	0	0	0	0	37
$r = 11$	19	6	4	1	0	1	0	36
$r = 12$	18	7	6	0	0	0	0	36
$r = 13$	20	6	3	1	1	0	0	36
$r = 14$	14	11	4	2	0	0	0	36
$r = 15$	17	9	3	1	1	0	0	36

TABLE IV
NUMBER OF NECESSARY REGISTERS FOR EXPERIMENT r.

which decide whether an incoming message is classified as "accepted" or "rejected". Looking at this tree, a programmer can already see important hints explaining failures of his SIP parser. In this case, the rules state that the message is likely to fail, if (i) the header field value in the "Event" header is set to "message-summary", or (ii) the "Error-Info" header exists, or (iii) the "Contact" parameter "transport" is included in the REGISTER message. On the other hand, if these three header fields are not included in the message and the "Via" parameter "rport" is included, it is likely that the message will be accepted by the VoIP proxy. The accuracy of this tree is already 91.43%.

Looking at the C4.5 decision tree derived at the end of the experiments 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. For example it gets quite clear that a SIP REGISTER message has to contain the header fields "CSeq" and "Call-ID" to be accepted from the VoIP proxy. On the other hand if the REGISTER message contains a "Replaces" header field it will most likely fail. Another observation that can be made is the fact that if a normal REGISTER message arrives that contains all necessary header fields, but additionally contains the "Warning" header field, it is likely that the message will fail. The same is true if the "Accept" header field contains an additional "level" parameter.

When comparing the first tree to the end tree, differences appear. For example the header field value "presence" of the header field "Allow-Events" does not even appear in the second tree anymore. On the other hand the header field "Error-Info" now decides whether a message will be accepted or rejected. This fact of course shows that over time the tree gets more and more precise, and wrong assumptions at an early stage of training indeed are removed over time.

VI. 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.

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 only REGISTER messages), and learns which REGISTER messages were accepted by this proxy, and which were not. Over time, Babel-SIP is able to accurately guess whether an incoming REGISTER messages is likely to be accepted by its proxy. If not, Babel-SIP changes the message such that the probability for acceptance is increased. By carrying out numerous experiments, we have demonstrated that with our approach the number of rejected messages is almost halved, and that Babel-SIP indeed gains knowledge over time and improves its effectiveness.

In the near future we will focus on analyzing INVITE messages, and creating semantic rules for changing header information. Since we regard our approach to be independent of SIP, 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] T. Abbes, A. Bouhoula, and M. Rusinowitch, "Protocol Analysis in Intrusion Detection Using Decision Trees," in *International Conference on Information Technology: Coding and Computing (ITCC'04)*, 2004, pp. 404–408.

[2] H. Abdelnur, R. State, and O. Festor, "KiF: A Stateful SIP Fuzzer," in *1st Int. Conference on Principles, Systems and Applications of IP Telecommunications*. iptcomm.org, 2007.

[3] A. Acharya, X. Wand, C. Wrigth, N. Banerjee, and B. Sengupta, "Real-time Monitoring of SIP Infrastructure Using Message Classification," in *MineNet'07*, 2007, pp. 45–50.

[4] B. Aichernig, B. Peischl, M. Weiglhofer, and F. Wotawa, "Protocol Conformance Testing a SIP Registrar: an Industrial Application of Formal Methods," in *5th IEEE Int. Conference on Software Engineering and Formal Methods*, 2007, pp. 215–224.

[5] S. Heisig and S. Moyle, "Using Model Trees to Characterize Computer Resource Usage," in *1st ACM SIGSOFT Workshop on Self-Managed Systems*, 2004, pp. 80–84.

[6] H. Kang, Z. Zhang, S. Ranjan, and A. Nucci, "SIP-based VoIP Traffic Behavior Profiling and Its Applications," in *MineNet'07*, 2007, pp. 39–44.

[7] T. Mitchell, *Machine Learning*. Mc-Graw-Hill, 1997.

[8] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," RFC 3261, June 2002.

[9] D. Wilking and T. Röfer, "Realtime Object Recognition Using Decision Tree Learning," in *RoboCup 2004: Robot World Cup VII*. Springer, 2005, pp. 556–563.

[10] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.