Yes We Can:
Uncovering Spoken Phrases in Encrypted VoIP Conversations

Goran Doychev Dominik Feld Jonas Eckhardt Stephan Neumann

May 28, 2009

Abstract

The growing importance of VoIP telephony over untrusted networks raises the requirements to encrypt VoIP calls. To achieve a good trade-off between audio quality and network traffic, Variable Bit Rate (VBR) codecs are widely employed. VBR codecs encode speech data at different bit rates depending on the complexity of the input signal. We implemented a practical side channel attack on VoIP applications using VBR codecs associated with length preserving encryption[1].

1 Introduction

Voice over IP (VoIP) becomes more and more important as an alternative to traditional telephony. As VoIP packets are transmitted over potentially insecure networks, it is well understood that VoIP packets must be encrypted to ensure confidentiality [Pro04]. Yet, applying common encryption techniques in a straightforward manner results in information leakage that can be exploited by side channel attacks.

Based on the works of Wright et al. [WBMM] [WBC$^+$08], we develop a practical side channel attack against encrypted VoIP conversations, using common VoIP clients such as Skype or Twinkle. Following their approach, we take advantage of the correlation between input speech and the resulting compressed data using Variable Bit Rate (VBR) encodings. If the VoIP application additionally uses a length preserving cipher, we expose an information leakage of the input speech. In particular, our attack allows us to detect given target phrases within an encrypted VoIP conversation. This is performed in two steps. First, a profile Hidden Markov Model (HMM) is trained with several encrypted transmissions of a certain sentence. After the training is completed, the attack can identify this sentence in an encrypted VoIP packet sequence.

[1]The ideas are based on the work *Spot me if you can : Uncovering spoken phrases in encrypted VoIP conversations* by Wright et al. [WBC$^+$08].

1.1 Related Work

In this section, we provide an overview on related research ideas, i.e. inference of sensitive information from encrypted and/or anonymized network connections.

The theoretical basis of the in hand work is described in the article by Wright et al. [WBC$^+$08]. We will immerse ourselves in their fundamental concept of profile Hidden Markov Models, an extension of Hidden Markov Models, in the following sections. Hidden Markov Models are widely used and well studied. For instance, the application of Hidden Markov Models is a standard technique used by the speech recognition community (cf. [WRLG90] [RRRG89]). Moreover, Hidden Markov Models have been successfully used to carry out side channel attacks. In [KW03], Karlof and Wagner introduced Input Driven Hidden Markov Models (IDHMM). IDHMMs are a modification of HMMs which are able to handle input. They developed an efficient algorithm to solve the input inference problem of IDHMMs. Karlof and Wagner used IDHMMs to model randomized countermeasures used in cryptographic devices that depend on an internally stored secret key.

In [WCJ05], Wang et al. present an attack on VoIP users' privacy. The attack uses a watermark technique which allows to identify and correlate encrypted VoIP calls in low latency anonymizing networks. A time adjustment of a few milliseconds can transform secure VoIP traffic to highly identifiable network traffic, i.e. the watermark is preserved in low latency anonymizing networks.

Verscheure et al. [VVA$^+$06] studied the feasibility of revealing pairs of anonymous conversing parties by exploiting the vulnerabilities inherent in VoIP systems. They validated their method on a very large standard corpus of conversational speech, and obtain pairing accuracy that reaches 95% after 5 minutes of voice conversation.

Wright et al. [WBMM] presented in an attack on encrypted VoIP calls which enabled them to infer the spoken language. They achieved an accuracy of more than 90% for 14 out of 21 languages and an overall binary classification of 86,6%. Their work exploits, just like their later work, the coding of speech at variable bitrates.

More general attacks on sensitive information have been proposed by Sun et al. [SSW$^+$02], where they present an attack on the non-identifiability of encrypted World Wide Web traffic based on HTTP object count and sizes. They investigated the occurring traffic requesting about 100.000 web pages. The count and size of objects contained in the webpages revealed all information needed to successfully exploit the encrypted web traffic in a straightforward manner. They stated that even with standard padding techniques encrypted web traffic remains still vulnerable.

In [FS00], the authors describe a class of cache-based timing attacks that can compromise the privacy of the users' web-browsing history. Depending on the cache state, the time it takes to perform different operations can vary. Malicious web servers can exploit this timing information of users' web browsers to reveal their past activities. The presented attacks are especially harmful since they are not based on implementation details but on basic properties of web browsers. This attack can be carried out unnoticed by the user.

Similar to the approach of Wright et al., we will exploit the speech coding of encrypted VoIP calls at variable bit rates. We borrow the idea of using profile Hidden Markov Models to recognize spoken phrases out of a set of training data.

1.2 Overview

Section 2 presents different VoIP protocols and codecs. Section 3 comprises the applications that are used to collect the VoIP packets and a visualization of the transmitted VoIP packets. Section 4 presents the concrete scenario of the attack. Section 5 shows the concept of the actual attack, an introduction to profile HMMs and the concrete implementation. Section 6 concludes this paper, interpreting the results of the attack and future research ideas are presented in section 7.

2 VoIP Protocols and Codecs

In this section we provide a short introduction to VoIP protocols and VoIP codecs. Voice over IP is a technique used to transmit speech data over IP networks. Apart from the most popular VoIP client Skype, which is proprietary, there exists a large number of VoIP open source implementations using open source protocols and codecs. These implementations differ in general in the protocols used to establish connections (Signaling Protocols), and in the codecs that are used to encode speech data in a bandwidth efficient manner (e.g. Linear Predictive Coding).

2.1 Signaling Protocols

Initiation and termination of VoIP connections need to be handled by protocols that are independent of speech streams. Apart from that fact, the availability and the IP address of the called user needs to be known within the network in order to start a VoIP call. Among the most popular signaling protocols are SIP and IAX.

SIP The Session Initiation Protocol (SIP) is an application layer protocol, which works independently of the underlying transport layer protocols, such as TCP or UDP. After the initial connection to a server over SIP[2], the protocol relies on the Real-time Transport Protocol (RTP) over UDP to transmit multimedia data like speech and video. The port to be used for the RTP protocol is determined in the SIP call signaling message. SIP has been standardized and governed primarily by the Internet Engineering Task Force (IETF).

IAX Another widely used signaling protocol is the Inter-Asterisk eXchange (IAX) protocol. In contrast to SIP, the IAX protocol carries signaling and speech data on the same port. IAX uses a single UDP data stream (usually on port 4569) to communicate between endpoints, multiplexing signaling and media flow. IAX easily traverses firewalls and network address translators. IAX supports trunking, multiplexing channels over a single link. When trunking, data from multiple calls are merged into a single stream of packets between two endpoints, thereby reducing the IP overhead without creating additional latency. This is advantageous in VoIP transmissions, in which IP headers use a large percentage of bandwidth.

[2]The initial connection to SIP servers is typically carried out over port 5060 or 5061.

2.2 Linear Predictive Coding

One of the central points of VoIP implementations is the quality of transmitted speech data under restricted bandwidth availability. There exists a wide range of audio codecs. Popular general purpose compression codecs are the MPEG-1 Layer III (MP3) codec and the RealAudio codec. Apart from these standard codecs, there exist audio codecs optimized for music coding or speech coding. In this section we will restrict our attention to speech coding algorithms as they form the basis of VoIP communication.

A widely used speech analysis technique is the Linear Predictive Coding (LPC). The history of LPC reaches back to 1966, when S. Saito and F. Itakura of NTT described an approach to automatic phoneme discrimination that involved the first maximum likelihood approach to speech coding. The LPC technique is based on the Linear Predicition (LP) method.

The linear correlation of the input signal's sampling values is used to predict probable upcoming input signals. Thereafter, the difference between the estimated signal and the real signal is calculated and transmitted. Apart from the described method, there exists an adaptive variant of LPC. Within the adaptive variant, prediction filters are chosen according to the input signal, which leads to more exact prediction. Apart from signal differences, details about the used prediction filter have to be transmitted. LPC is a low bit rate coding technique.

Many coding techniques are based on the idea of LPC. We will present two of them in more detail in the following.

Code Excited Linear Predicition One of the oldest speech codecs is the Code Excited Linear Predicition (CELP) codec. It has been introduced in 1985 by Schroeder and Atal [SA85]. They propose a way to code the residue signal of the linear prediction in an resource efficient way in order to keep the bit rates as low as possible. The central points of the CELP technique can be summarized in four steps.

- Apply the linear prediction filter to the input signal.
- Use an adaptive and a fixed codebook as the input (excitation) of the LP model.
- Perform a search in closed-loop in a perceptually weighted domain.
- Apply vector quantization.

Due to the very limited bandwidth of mobile telephony, CELP and its variants have become the main speech codec for mobile phones.

Speex Based on the larger bandwidth of internet connections, VoIP requires more adequate speech codecs than the low bit rate CELP codecs. The defined goals of the Speex codec are to guarantee high quality speech at bandwidth efficient resource usage.

Speex supports three different sampling rates in order to provide the most adequate speech transmission depending on the available bandwidth: 32kHz, 16kHz or 8kHz. Apart from the adaptive sampling rate provided by the codec, it supports VBR encoding. VBR allows the codec to adapt its bit rate depending on the complexity of the audio stream. For all available bit rates the encoding of audio stream is carried out with an adapted linear prediction filter as the CELP filters. For example, vowels and high-energy transients need to be encoded at higher bit rates while fricatives can be encoded at lower bit rates to obtain the same quality. In practice, VBR coding provides high quality speech with efficient bandwidth usage.

In addition to the above mentioned adaptive sampling and bit rate, Speex provides Average Bit Rate coding, Voice Activity Detection, Discontinuous Transmission, and many more. We refer the interested reader to the Speex project webpage [spe05].

3 Applications

In this section, the tools and scripts that have been used to carry out the implemented attack are presented.

3.1 Network Monitoring

In order to carry out the VoIP packet size analysis, VoIP packets need to be filtered out of the network traffic. We created a command line script that relies on the well known Linux packet sniffer tcpdump [tcp]. During a VoIP conversation, we use tcpdump to monitor the network traffic and filter out the packets of interest. Finally, important information on these packets, i.e. time and packet size, is written to an output file and is used as input of the splitter script as described below.

3.2 Splitter

The training of profile HMMs and the preparation of phrase pools require large sequences of packet sizes from spoken phrases. In our setup, we transmitted different phrases several hundred times. We had to split the resulting packet size sequence in sequences corresponding to one transmission of the phrase. We created a script which supports us to generate such sequences and eliminates the pauses between two subsequent phrases.

3.3 Visualization

Since we started our work by considering several VoIP programs possibly using encryption and variable bitrate audio coding, we first had to assure that the selected tools had these features indeed. Concerning information about software, especially about programs of a lower level of awareness, the internet often offers no more than incomplete and sometimes even incorrect statements. To solve these problems, we implemented a tool that visualizes the packet sizes that are transmitted during a VoIP call.

Our visualization tool simply fulfils the most needed requirements to roughly analyze and compare network traffic. After a network protocol file has been selected, it retrieves size and time from each packet displays them as graph scaled by packet number or by time. Apart from easily being able to recognize a variable bitrate, one can also compare different network traffic sequences, as our visualization tool can administrate an arbitrary number of them and display several at a time. As one would assume, we were able to confirm that the same sentence, even if it was spoken from a different person, results in a graph containing many similarities (see Figure 1).

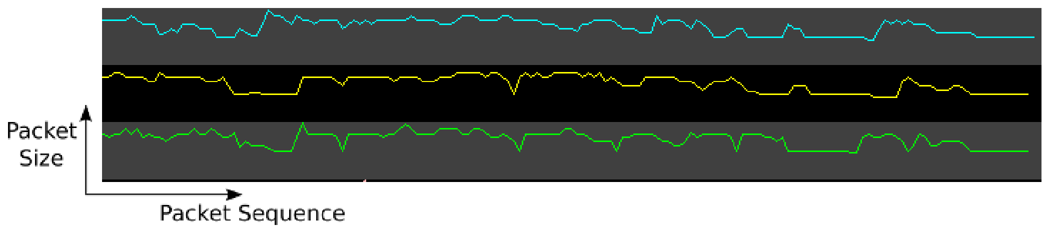

Figure 1: Visualization of Three Utterances of the Same Sentence.

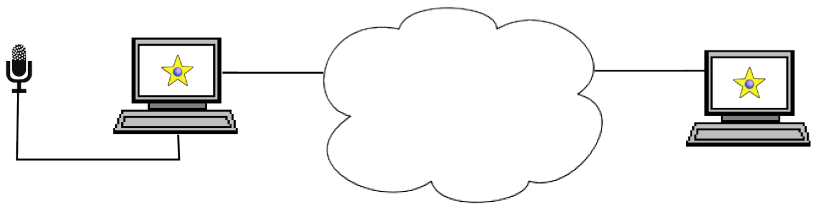

Figure 2: Setup of the Experiment.

The visualization was certainly a helpful tool for validating the correctness of our approach by providing visual feedback.

4 Experimental Setup

In order to find a reasonable VoIP application that could be used to realize and validate our attack, we had to take into account several facts. As the basic idea is to exploit the encoding of speech at variable bit rates, an adequate codec had to be chosen. As mentioned above, the Speex codec is a widely used VBR encoding algorithm. Moreover, the VoIP application of investigation has to support length preserving encryption such as AES.

Twinke Twinkle [twi] is an open source VoIP application which uses the SIP signaling protocol. Apart from the Speex codec it supports many other audio codecs, e.g. GSM, iLBC and G.726. Encryption of speech data is carried out over an extended version of the RTP protocol, namely the ZRTP protocol. ZRTP extends the original protocol in a way, that at the beginning of a call a symmetric key is exchanged with help of the Diffie-Hellman key exchange protocol. Afterwards, the RTP packets are encrypted with AES-128, or AES-256.

Hardware Setup The goal of this work is to construct an attack that works under real world circumstances. In order to carry out our attack, we used two computers at different locations. We started an encrypted Twinkle conversation between those two endpoints over the Internet (cf. Figure 2).

After the conversation has been setup, we transmitted four spoken phrases out of the audiobook “Alice in Wonderland” from Project Gutenberg [gut], a collection of books under the public domain, each of them 40 times. The four phrases are:

1. *Flamingoes and mustard both bite. And the moral of that is: Birds of a feather flock together.*

2. *Have you seen the Mock Turtle yet? No, said Alice. I don't even know what a Mock Turtle is. It's the thing Mock Turtle Soup is made from, said the Queen.*
3. *Once, said the Mock Turtle at last, with a deep sigh, I was a real Turtle.*
4. *Why did you call him Tortoise, if he wasn't one? Alice asked. We called him Tortoise because he taught us.*

After the transmission of the four phrases, phrase 1 was spoken 40 times by two of our members. In addition to the phrase 1 to 4 spoken by a female voice with British accent, this time phrase 1 was spoken by a male German native speaker and a male Bulgarian native speaker in order to extend the results to be more realistic.

The outgoing traffic of the call initiating system was captured with the help of our network monitoring script. The resulting output files of each phrase were splitted with the above mentioned splitter script. Finally, this yields to five files, each of them containing the packet sizes and timestamps of 40 transmissions of one phrase. The first 30 transmissions were used to train the profile HMMs, while the last ten transmissions were used in order to carry out the attack. The model was trained with 100 iterations of the Baum-Welch training algorithm, while the intermediate results after each iteration were serialized in order to analyze the correlation between the number of iterations and the accuracy of the attack. The attack was executed using each of the 100 trained models.

5 The Attack

Neither the theory of (Profile) Hidden Markov Models nor the application to speech recognition is new [Rab90]. Baum and his collegues published the basic theory in a series of classic papers [BP66] [BJ67] [BR68] [BPSW70] [Bau72]. In this section, we will first give some basic information about the complex task of speech processing and speech coding followed by a short introduction to Profile Hidden Markov Models and their application in our attack.

5.1 Correlation Between Speech and Packets

One of the main tasks of every VoIP program is to transmit audio signals over the network to another client. Therefore analogue audio data is recorded by microphone and converted into a digital representation which allows both parties to recover the original data. Furthermore, the network traffic should be minimized since network resources are typically limited and preferred to be held as low as possible. Yet one wishes simultaneously to ensure a certain quality of the audio data. On that account VoIP clients come with a built in source coding technique.

The broad range of audio coding techniques can be divided into those which use a fixed bitrate and the ones which use a variable one. A variable bitrate has the advantage that one can usually achieve a way better audio quality while producing digital data of the same size as with a fixed bitrate. And vice versa one can ensure the same audio quality as with a fixed bitrate while keeping the quantity of data lower.

To obtain these benefits one leverages that audio signals have a large variation in terms of complexity. In particular silence, which accounts for about 63% of a usual conversation for each of the parties [WBC$^+$08], can be coded with fewer bits than fricatives without human beings possibly noticing any difference. Fricatives on the other hand can be coded with fewer bits than vowels without any noticeable quality loss. In our work we only focus on coding methods using variable bitrate, since we want to exploit the fact that one can draw conclusions from the resulting bitrate about what was coded in the first place.

It is indeed crucial to get familiar with the process of digitalisation and coding of audio data for understanding why this side channel attack can be successfully carried out, and therefore a short introduction follows.

Transforming a continuous amplitude signal into a digital representation with the minimal distortion is called quantization. Single and multiple signal values are converted into respective scalar and vector quantizations using classical pattern recognition techniques [Rub94] [RJ93]. Each of the resulting vector quantizers is assigned a code word in a codebook where each code word is representative for an index in the codebook.

A variable bitrate coding usually allows a modulation of several discrete bitrates. For each of them an appropriate codebook is used which generally has $2^{bitrate}$ entries to maximize the number of possible distinct audio signal representations and hence minimize the distortion. Consequently, each audio sample is coded with a number of bits equal to the current bitrate. Increasing the bitrate obviously increases the size of the codebook and thus offers a far better approximation for each audio signal.

Concerning speech coding some phonemes require a higher bitrate than others in order to have users not notice any difference in quality during the conversation. Seeing this from a different perspective, just by observing the bitrates used to code the conversation one may extract some information about what was actually said (see Figure 3).

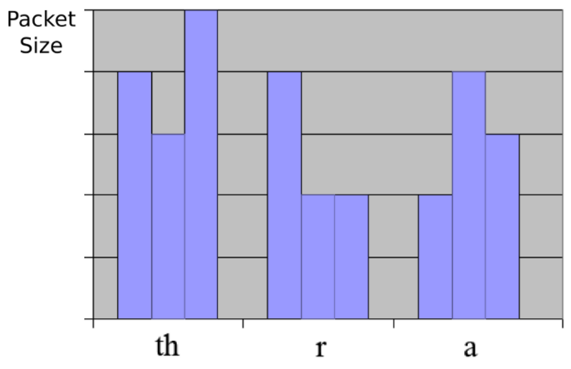

Figure 3: Encoding of three Phonemes, each of them encoded into three consecutive Packets.

In common VoIP tools the resulting code words are not sent immediately over the network, but instead buffered first and eventually encrypted and sent in intervals of about 20ms. Note that we perform a side channel attack on VoIP software meaning that the actual encryption algorithm used does not matter to us, since we only consider packet sizes which are unaffected by the encryption. The only thing we need to ensure is that the encryption used is length preserving so packet sizes really do not change due to encryption. However this is usually the case and thus no restriction to the software we analyse.

So, instead of seeing the individual bitrates, all we can observe are the packet sizes. Even though the entropy of each packet size is lower than the entropy of the bitrates contained in each packet it still reveals a lot of information about the initial speech.

Additionally, we do not observe the packets separate but rather as part of a sequence of packets that belong to a word, a phrase or even a whole sentence. These longer sequences offer a lot of distinguishing marks meaning that observing a packet sequence that fulfils these distinguishing marks for the most part gives certainty that with a high probability it really was the expected sentence.

Recording and coding a given sentence several times will undeniably lead to different packet sequences, especially if the sentence is spoken by different people. Yet a great number of the main characteristics of the packet sequences will be present in most of the observations: In particular, pauses between words resulting in one or several small packets and the alternation of smaller and larger packets depending on the occurring phoneme in the respective words. One can see the correlation between speech and the resulting sequence of packet sizes in Figure 4 and Figure 5.

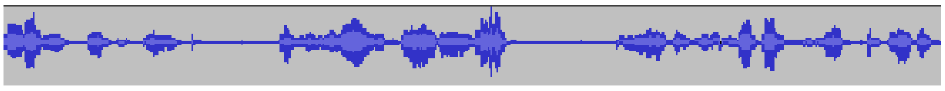

Figure 4: Speech Recording of sentence one.

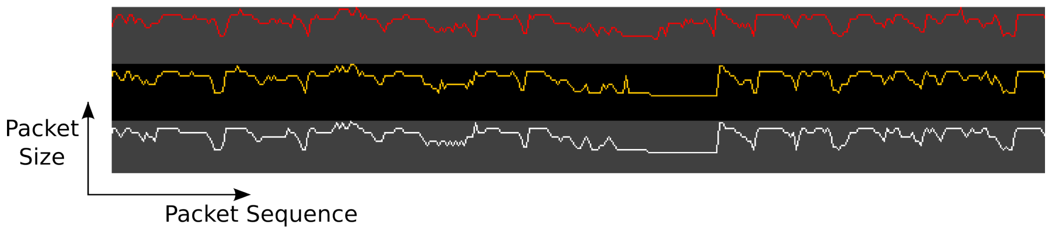

Figure 5: Three resulting packet sequences of sentence one.

To evaluate collected data, i.e. comparing packet sequences to decide if one recognizes a known sentence or phrase, a commonly used technique to solve problems in this vein is modelling them as profile Hidden Markov Models. Some basic knowledge about this method is to be conveyed in the next section.

5.2 Introduction to (Profile) HMMs

Hidden Markov Models (HMMs) are especially known for their application in temporal pattern recognition such as speech [BCG$^+$90], face [ZCPR03], handwriting [SB04], gesture [Ges00], music [SPMB02], part-of-speech tagging [TIM02] and bioinformatics. The beauty of a HMM is that it is able to reveal the underlying process of signal generation even though the properties of the signal source remain greatly unknown [DFL05].

Hidden Markov Model

A HMM is composed of a number of interconnected *states*, each of which emits an observable output symbol (packet size). Each state has two kinds of parameters. *Symbol emission probabilities* are the probabilities of emitting each possible symbol from a state. *State transition probabilities* are the probabilities of moving from the current state to a new state. A *sequence* is generated by starting at an initial state and moving from state to state until a termination state is reached, emitting symbols from each state that is passed through [Edd95].

Formally, a Hidden Markov Model is defined as a quintuple $\lambda = (S, A, B, \pi, V)$ with

- $S := \{s_1, \dots, s_n\}$ – the set of all states.
- $A := \{a_{ij}\}$ – the matrix of state transition probabilities.
- $B := \{b_1, \dots, b_n\}$ – the set of output (observable) symbols.
- π – the initial probability distribution.
- V – the matrix of emission probabilities.

There are three basic problems of interest for the model to be useful in real-world applications. These problems are the following:

Problem 1 is the evaluating problem. Given a model and a sequence of observations, how do we compute the probability that the observed sequence was produced by the model. This problem can also be viewed as how well a given model matches a given observation. In our context, solving this problem tells us whether a given sequence of observations (sequence of packets) corresponds to a given sentence or not.

Given the observation sequence $O := O_1 O_2 \dots O_\tau$ and the model $\lambda = (S, A, B, \pi, V)$. How do we efficiently compute $P(O \mid \lambda)$, the probability of the observation sequence, given the model λ?

Problem 2 tries to uncover the sequence of hidden states of the model, i.e. to find the *correct* state sequence. There is no *correct* state sequence, in general, thus an optimality criterion is used to solve this problem as best possible, in most practical solutions.

Given the observation sequence $O := O_1 O_2 \dots O_\tau$ and the model $\lambda = (S, A, B, \pi, V)$. How do we choose a corresponding state sequence $Q = q_1 q_2 \dots q_\tau$ which is optimal in some meaningful sense?

Problem 3 is the one which attempts to optimize the model parameters. The observation sequence used to adjust the model is called a training sequence since it is used to *train* the HMM. The training problem is crucial for most applications of HMMs, since it allows to optimally adapt model parameters to observed training data. Baum and Welch proposed one solution to this problem in [BPSW70]. In our context, we use the training algorithm from Baum and Welch to train our model of the search sentence based on a set of training sequences.

Given the observation sequence $O := O_1 O_2 \dots O_\tau$ and the model $\lambda = (S, A, B, \pi, V)$. How do we adjust the model parameters λ to maximize $P(O \mid \lambda)$?

Example 1: Hidden Markov Model and Speech Recognition

Let $\lambda := (S, A, B, \pi, V)$, with $S := \{a, b, c\}$ be the set of spoken phonemes and

$$B := \{o_b := big, o_m := medium, o_s := small\}$$

be the set of observable outputs (packets). Initially, the state transition probability distribution, the emission probability distribution and the initial distribution are set to be uniform.

$$A = \frac{1}{3} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \text{ and } V = \frac{1}{3} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

First we train our model with a set of training sequences $TS \subseteq S \times S \times S$. In this case we require the training sequences to be of length 3 and to be only composed of symbols from S. As result, we get the state altered transition matrix A and the output transition matrix V.

$$A = \begin{pmatrix} 0.4 & 0.3 & 0.3 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{pmatrix} \text{ and } V = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

Figure 6 depicts the resulting HMM.

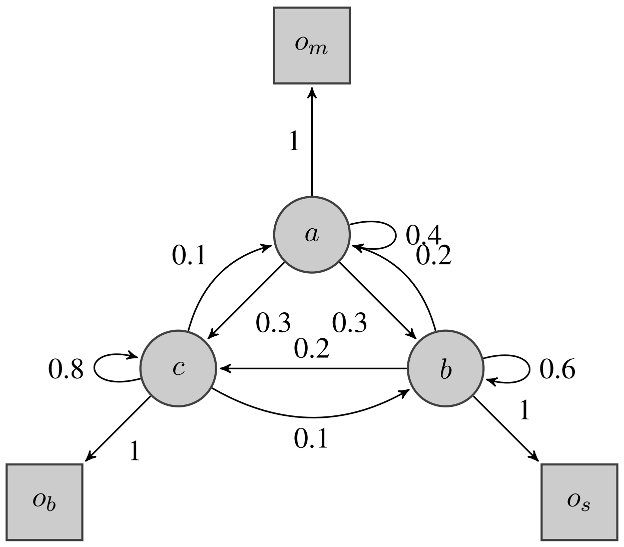

Figure 6: A Hidden Markov Model for the Example.

Given a sequence of observed outputs (packet sizes), we want to know the most probable state path producing this sequence. As one can see, in this simple example, there is a one-to-one correspondence between spoken phonemes and emitted packets.

This model is sufficient if we require the training sequences to be of the same length and to be only composed of symbols from S. But human speech is known to exhibit a high degree of variability, and the adaptive compression performed by the VoIP client may contribute additional variance to the resulting packet sizes [WBC$^+$08].

We need a model that can handle sequences that differ in their length, caused by insertions or deletions of several packets, due to the VoIP client or network transmission. To solve this, we can adapt *profile Hidden Markov Model* techniques [Edd95] to the task of finding words and phrases.

Profile Hidden Markov Model (Profile HMM)

A profile HMM consists of three interconnected chains of states (see Figure 7) describing the the expected packet lengths at each position in the sequence.

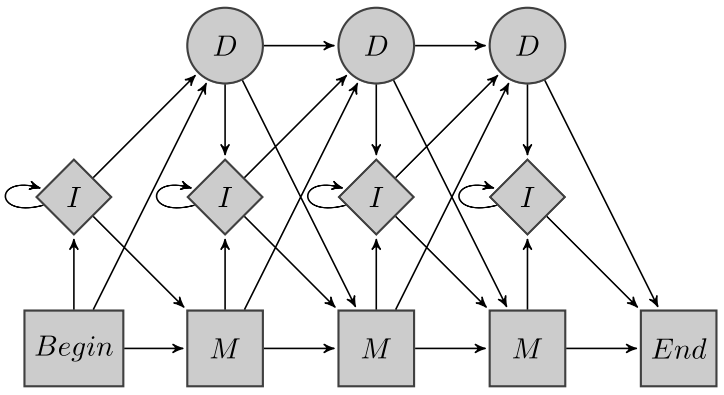

Figure 7: Profile HMM systematically depicted.

In our case, the *Match states*, depicted as squares, represent the expected distribution of packet sizes at each position in the sequence. *Insert states* and *Delete states*, shown as diamonds and circles, allow for variations from the typical sequence. For example, if a sequence contains additional packets or less packets than expected, the insert states *insert* additional packets in the expected sequence and the delete states *omit* packets from the expected sequence.

Using this approach, we set the match states probability to be uniform, initially. Given a set of example sequences, which in fact can now vary in their length, the profile HMM is trained using the Baum and Welch [BPSW70] training algorithm to iteratively improve the model's parameters to better represent the given training sequences.

5.3 Implementation of the Attack

In this section, we will focus on the implementation of our attack. First, an overview on existing HMM algorithms is provided followed by a description of the attack.

For our implementation we used the libraries provided by BioJava [bio]. BioJava is a mature open-source project that provides a framework for processing biological data. BioJava contains powerful analysis and statistical routines, tools for parsing common file formats and packages for manipulating sequences and 3D structures. It enables rapid bioinformatic application development in the Java programming language [bio]. It contains classes that model profile HMMs, the Baum-Welch algorithm and the Viterby Search algorithm among others. We chose BioJava for our implementation, since it is an well-documented open-source project, and offers the features we needed.

Alternatively, the algorithms could be implemented from scratch, which would give a greater degree of control in the used methods. Other available (profile) HMM algorithm collections are the library GHMM [ghm], the bioinformatics oriented software packages HMMER [hmm] and SAM [sam] and the speech recognition oriented software package HTK [htk].

The Implementation

The implementation consists of two phases. The first phase comprises the creation and the training of the model, and the second phase consists of the actual attack on several sentences.

The Training Phase addresses the creation and training of a profile HMM. In order to create the profile HMM, the set of observable symbols has to be defined (see Listing 1 line 1). The set of observable symbols consists of the packet sizes that have been transmitted, thus, for every packet size an observable symbol is created, if it was not created yet.

One further important issue is the number of match states (and the resulting number of insert and delete states) of the profile HMM. In our approach, the number of match states is computed as the average number of packets in the training sequences, since the average corresponds to the expected number of packets in the attack sentences. Our *PacketAlphabet* class computes that number while creating the set of observable symbols.

The profile HMM is represented by the BioJava *ProfileHMM* class. It is initialized with the set of observable symbols, the number of match states and two instances of *DistributionFactory.DEFAULT* (see Listing 1 line 2). The distribution factory is used to create the insert and the match states, and in our approach we use an uniform distribution for the transitions.

Finally, we create a dynamic programming matrix (*DP*) , an object that can perform dynamic programming operations upon sequences with HMMs. The *DP* is used to train our profile HMM.

Listing 1: Creation of the Profile HMM

```
1  _alphabet = new PacketAlphabet(_trainingsData);
2  _profileHMM = new ProfileHMM(_alphabet.getAlphabet(),_alphabet.getAveragePacketNumbers(),
       DistributionFactory.DEFAULT,DistributionFactory.DEFAULT,pHmmName);
3  _dynamicProgrammingMatrix = DPFactory.DEFAULT.createDP(_profileHMM);
```

Having created and initialized the profile HMM and the dynamic programming matrix, we first train the profile HMM with an *SimpleModelTrainer*, to cause the models' probabilities to be uniform (see Listing 2 line 1–6).

The Baum-Welch training algorithm is contained in the *BaumWelchTrainer* class. We train the model (see Listing 2 line 12) using a self made *StoppingCriteria*, that stops the algorithm after a certain amount of rounds have passed.

Listing 2: Training the Profile HMM

```
1  ModelTrainer mt = new SimpleModelTrainer();
2  //register the model to train
3  mt.registerModel(_profileHMM);
4  //as no other counts are being used the null weight will cause everything to be uniform
5  mt.setNullModelWeight(1.0);
6  mt.train();
7
8  //create a BW trainer for the dp matrix generated from the HMM
9  BaumWelchTrainer bwt = new BaumWelchTrainer(_dynamicProgrammingMatrix);
10
11 CylcleStoppingCriteria stopper = new CylcleStoppingCriteria(_prefix, _trainSaveModulus,
       numIter,this);
12 bwt.train(_alphabet.getTrainingSet(),1.0,stopper);
```

It is important to notice, that one can serialize[3] the profile HMM after each training round, and can run the attack based on each of these intermediate profile HMMs. This is especially useful when many training runs are needed or one wants to evaluate the necessary number of training runs.

[3]Serialization is the process of converting an object into a sequence of bits in order to be stored on a storage medium or to be transferred over the network.

The Attack Phase takes a sequence of packets, and computes the probability of the most likely path producing that specific sequence of packets as output in two ways. First, the probability under the *Null Model* is computed (see Listing 3 line 1), and then the probability under the *Learned Model* is computed (see Listing 3 line 2). The first probability states whether a randomly chosen profile could have produced the output, and the second probability states whether the learned model could have produced the output. Finally, the ratio of these two probabilities is returned yielding the matching probability of the model.

Listing 3: The Attack.

```
1 StatePath p1 = _dynamicProgrammingMatrix.viterbi(sequence,ScoreType.NULL_MODEL);
2 StatePath p2 = _dynamicProgrammingMatrix.viterbi(sequence,ScoreType.ODDS);
3
4 return p2.getScore() / p1.getScore();
```

6 Results and Interpretation

In our expirement, we trained a profile HMM for each of the five phrases mentioned in Section 4. To attack a particular phrase, ten obtained packet sequences of that phrase were matched with each of the five profile HMMs yielding ten matching probabilities for each of the five models. We calculated the mean value of the ten matching probabilities and used this value to decide which profile HMM matches the phrase.

The obtained results show that our attack can identify the spoken sentences in all cases by comparing the matching probability of the trained profile HMMs.

Figure 8 shows the result of the attack, which tests ten different packet sequences of the spoken sentence (sentence one) with the five trained HMMs. Figure 9 depicts the result of the attack, where ten different transmissions of sentence two were tested with the five trained HMMs.

The number of training iterations are displayed on the x-axis of the graph, and the mean of the matching probabilities are displayed on the y-axis. The red dots correspond to the HMM trained with transmissions of sentence one; green dots corresponds to sentence two; blue to sentence three, and yellow to sentence four. The black dots corresponds to the HMM trained with the spoken transmissions of sentence one.

One can observe in Figure 8 and Figure 9 that the matching probability of the correct HMM is in all cases greater than the matching probability of the other HMMs. Even after a few training iterations, the correct sentence can be discovered by comparing the matching probabilities of all HMMs. This means that our attack can identify sentences out of a given set during an encrypted VoIP conversation.

Interestingly, in Figure 8 the matching probabilities of all HMMs grow with the number of iterations, and the difference between the matching probabilities of the correct HMM and the others remains nearly constant. In contrast, in Figure 9 the matching probabilities form a cyclic structure, while the difference between the matching probabilities of the correct model and the others grow.

Furthermore, it is important to notice, that the HMM trained with transmissions of the spoken sentence does not identify the recorded version of that sentence. But there is a fundamental difference between the recorded sentence and the spoken one: The spoken sentence was uttered by male German and Bulgarian native speakers respectively, while the recorded sentence was spoken by a female British English speaker. This could be extended to a further topic of investigation.

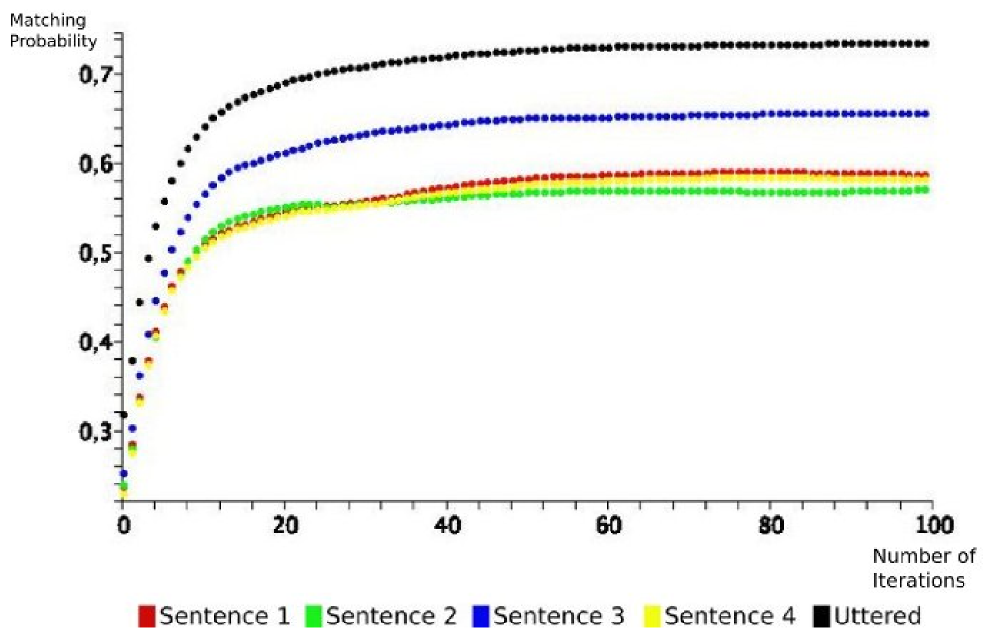

Figure 8: Attack on the Uttered Sentence.

We did not find an optimal number of iterations of the Baum-Welch training algorithm. Nevertheless, the attack performs well. It is able to identify the correct phrase in all cases.

7 Future Work

The obtained results are promising and show that it is possible in general to recover words or phrases in encrypted VoIP calls. Nevertheless, our work still forms a skeletal structure for more sophisticated attacks against encrypted VoIP calls. This section is dedicated to possible future extension of our work.

As already proposed in [WBC$^+$08], it is possible to synthesize phrases out of given phonemes in order create a larger set of training data. Hence, a further work could exploit a set of spoken phrases S in order to generate a much larger set, namely a reasonable subset of the powerset of all phonemes appearing in S. The attack would no more be depending on pre determined languages and could theoretically recover all words of an encrypted call [4].

Apart from the leakage of complete words or phrases of encrypted VoIP calls, an eavesdropper can already do harm by knowing other facts than the spoken words. Therefore, future works could investigate if it is possible to obtain information about the age, the gender, or the language of the speaker (cf. [WBMM]). In addition, the speech pattern of individuals could possibly leak enough information to identify the speakers within an encrypted VoIP call.

Moreover, the robustness of our implementation needs to be investigated further. We propose the comparison of disturbed phrases, i.e. additional noise, distorted voices, etc. By now, we are not aware of their influence on the results.

[4]We emphasize that this is hardly possible in practice as the probability between matching and nonmatching words would get very close.

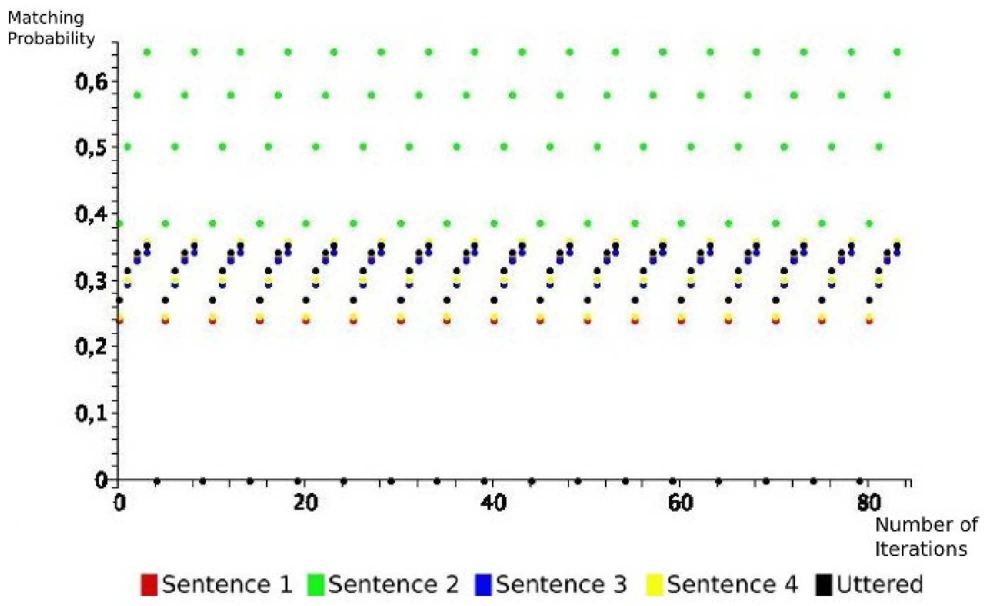

Figure 9: Attack on Sentence Two.

References

[Bau72] Leonard E. Baum. An inequality and associated maximation technique in statistical estimation for probabilistic functions of Markov processes. *Inequalites*, 3:1–8, 1972.

[BCG$^+$90] Paul Bamberg, Yen-lu Chow, Laurence Gillick, Robert Roth, and Dean Sturtevant. The Dragon continuous speech recognition system: a real-time implementation. In *HLT '90: Proceedings of the workshop on Speech and Natural Language*, pages 78–81, Morristown, NJ, USA, 1990. Association for Computational Linguistics.

[bio] Bio Java. `http://biojava.org/wiki/Main_Page`. [Online; accessed 29-April-2009].

[BJ67] Leonard E. Baum and Egon J.A. An inequality with applications to statistical estimation for probabilistic functions of Markov processes and to a model for ecology. *Bull. Amer. Math. Soc.*, 73:360–363, 1967.

[BP66] Leonard E. Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state Markov chains. *The Annals of Mathematical Statistics*, 37(6):1554–1563, 1966.

[BPSW70] Leonard E. Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970.

[BR68] Leonard E. Baum and Sell G. R. Growth functions for transformations on manifolds. *Pac. J. Math*, 27(2):211–227, 1968.

[DFL05] Liang Dong, Say Wei Foo, and Yong Lian. A two-channel training algorithm for Hidden Markov Model and its application to lip reading. *EURASIP J. Appl. Signal Process.*, 2005:1382–1399, 2005.

[Edd95] S. Eddy. Multiple alignment using Hidden Markov Models. *Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology*, pages 114–120, 1995.

[FS00] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *CCS '00: Proceedings of the 7th ACM conference on Computer and communications security*, pages 25–32, New York, NY, USA, 2000. ACM.

[Ges00] An HMM-based approach for gesture segmentation and recognition. In *ICPR '00: Proceedings of the International Conference on Pattern Recognition*, page 3683, Washington, DC, USA, 2000. IEEE Computer Society.

[ghm] GHMM. `http://ghmm.sourceforge.net`. [Online; accessed 29-April-2009].

[gut] Project Gutenberg. `http://www.projectgutenberg.com`. [Online; accessed 29-April-2009].

[hmm] HMMER. `http://hmmer.janelia.org`. [Online; accessed 29-April-2009].

[htk] HTK. `http://htk.eng.cam.ac.uk`. [Online; accessed 29-April-2009].

[KW03] Chris Karlof and David Wagner. Hidden Markov Model cryptanalysis. Technical Report UCB/CSD-03-1244, EECS Department, University of California, Berkeley, 2003.

[Pro04] N. Provos. Voice over misconfigured internet telephones. `http://vomit.xtdnet.nl/`, 2004. [Online; accessed 15-April-2009].

[Rab90] Lawrence R. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. pages 267–296, 1990.

[RJ93] Lawrence Rabiner and Biing-Hwang Juang. *Fundamentals of speech recognition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[RRRG89] R. Rohlicek, W. Russell, S. Roukos, and H. Gish. Continuous Hidden Markov modeling for speaker-independent word spotting. pages 627–630, 1989.

[Rub94] Jeffrey Rubin. *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[SA85] M. Schroeder and B. Atal. Code excited linear prediction (CELP): high quality speech at very low bit rates. pages 937–940, 1985.

[sam] SAM. `http://compbio.soe.ucsc.edu/sam.html`. [Online; accessed 29-April-2009].

[SB04] Andreas Schlapbach and Horst Bunke. Off-line handwriting identification using HMM based recognizers. In *ICPR '04: Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 2*, pages 654–658, Washington, DC, USA, 2004. IEEE Computer Society.

[spe05] The Speex project page. http://www.speex.org, 2005.

[SPMB02] Jonah Shifrin, Bryan Pardo, Colin Meek, and William Birmingham. HMM-based musical query retrieval. In *JCDL '02: Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, pages 295–300, New York, NY, USA, 2002. ACM.

[SSW$^+$02] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. Statistical identification of encrypted web browsing traffic. In *IEEE Symposium on Security and Privacy*. Society Press, 2002.

[tcp] TCPdump. `http://www.tcpdump.org`. [Online; accessed 29-April-2009].

[TIM02] Kristina Toutanova, H. Tolga Ilhan, and Christopher D. Manning. Extensions to HMM-based statistical word alignment models. In *EMNLP '02: Proceedings of the ACL-02 conference on Empirical methods in natural language processing*, pages 87–94, Morristown, NJ, USA, 2002. Association for Computational Linguistics.

[twi] Twinkle phone. `http://www.twinklephone.com`. [Online; accessed 29-April-2009].

[VVA$^+$06] Olivier Verscheure, Michail Vlachos, Aris Anagnostopoulos, Pascal Frossard, Eric Bouillet, and Philip S. Yu. Finding "who is talking to whom" in VoIP networks via progressive stream clustering. *Data Mining, IEEE International Conference on*, 0:667–677, 2006.

[WBC$^+$08] Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M. Masson. Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 35–49, Washington, DC, USA, 2008. IEEE Computer Society.

[WBMM] Charles V. Wright, Lucas Ballard, Fabian Monrose, and Gerald M. Masson. Language identification of encrypted VoIP traffic: Alejandra y Roberto or Alice and Bob?, booktitle = SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, year = 2007, isbn = 111-333-5555-77-9, pages = 1–12, location = Boston, MA, publisher = USENIX Association, address = Berkeley, CA, USA,.

[WCJ05] Xinyuan Wang, Shiping Chen, and Sushil Jajodia. Tracking anonymous peer-to-peer VoIP calls on the internet. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 81–91, New York, NY, USA, 2005. ACM.

[WRLG90] J. G. Wilpon, L. R. Rabiner, C. H. Lee, and E. R. Goldman. Automatic recognition of keywords in unconstrained speech using Hidden Markov Models. *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on*, 38(11):1870–1878, 1990.

[ZCPR03] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld. Face recognition: A literature survey. *ACM Comput. Surv.*, 35(4):399–458, 2003.

A Sources

Listing 4: Main.java

```java

package de.voipattack;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

import org.biojava.bio.BioException;
import org.biojava.bio.dp.StatePath;
import org.biojava.utils.ChangeVetoException;

import de.voipattack.attack.Attack;
import de.voipattack.util.Logger;

public class Main {

    private static Attack _attack;

    public static void main(String[] args) {
        Logger.Log("Usage voipattack -t <traindatafile" +
                "> -s <serializationfile> -d <deserializationfile> -r <#training runs> -i
                    <save Interval>" +
                " -interactive -a <attackfile1,...,attackfileX>");

        String serializationFile = "";
        String deserializationFile = "";
        String traindataFile = "";
        int trainingRuns = 2;
        int saveInterval = 0;
        boolean interactive=true;
        String attackFiles[] = null;

        if(args.length > 0){
            interactive = false;
        }
        // First parse the arguments and decide whether the training data has to be read
            from
        // a textfile or a deserialization file...
        for (int i = 0; i < args.length; i++) {
            if (args[i].equals("-t")) {
                if (args.length > i+1)
                    traindataFile = args[i+1];
            } else if (args[i].equals("-s")) {
                if (args.length > i+1)
                    serializationFile = args[i+1];
            } else if (args[i].equals("-d")) {
                if (args.length > i+1)
                deserializationFile = args[i+1];
            }else if(args[i].equals("-r")) {
                try{
                    if (args.length > i+1)
                        trainingRuns = Integer.parseInt(args[i+1]);
                }catch(Exception e){
                    Logger.Log("Parameter -r " + args[i+1] + " ist kein Integer");
                }
            }else if(args[i].equals("-i")) {
                try{
                    if (args.length > i+1)
                        saveInterval = Integer.parseInt(args[i+1]);
                }catch(Exception e){
                    Logger.Log("Parameter -r " + args[i+1] + " ist kein Integer");
                }
            }else if (args[i].equals("-interactive")){
                interactive = true;
            }else if (args[i].equals("-a")){
                try{
                    if(args.length > i+1){
                        attackFiles = args[i+1].split(",");
                    }
                }catch(Exception e){
                    Logger.Log("attackFiles Parameter");
                }
            }
        }

        _attack = new Attack(serializationFile,saveInterval);

        if (deserializationFile.equals("") && interactive)
            deserializationFile = getLineFromIn("Read the model from file? Enter filename: ");

        if (!deserializationFile.equals("")) {
            File dFile = new File(deserializationFile);
            if (!dFile.exists())  {
                Logger.Log("<deserializationFile> : "  + deserializationFile + " does not exist!");
                System.exit(1);
            }
            _attack.deSerialize(new File(deserializationFile));
        }

        if (traindataFile.equals("") && interactive)
            traindataFile = getLineFromIn("Improve the model? Enter filename:");

        if (!traindataFile.equals("")) {
            File trainData = new File(traindataFile);

            if (!trainData.exists()) {
                Logger.Log("<traindatafile> : "  + traindataFile + " does not exist!");
                System.exit(1);
            }
            readTrainingDataFromFile(trainData);
            _attack.train(trainingRuns);

            if (serializationFile.equals("") && interactive)
                serializationFile = getLineFromIn("Save the intermediate result? Enter filename:");

            if (!serializationFile.equals("")) {
                File sf = new File(serializationFile);
                try {
                    if (sf.exists())
                        sf.delete();
                    sf.createNewFile();
                    _attack.serialize(sf);
                } catch (IOException e) {
                    Logger.Debug("Cannot create File", e);
                }
            }
        }

        Logger.Log("");
        int i = 0;
        while (true) {
            File attackData = null;
            String filename = "";
            if (attackFiles != null && attackFiles.length > i){
                filename = attackFiles[i];
                i++;
            } else if (attackFiles != null) {
                System.exit(0);
            } else {
                filename= getLineFromIn("Enter attack file or exit:");
            }

            if (filename.compareToIgnoreCase("exit") == 0)
                System.exit(0);

            if(filename != ""){
                attackData = new File(filename);
                attackFromFile(attackData);
            }
        }
    }

    private static void readTrainingDataFromFile(File theFile) {
        BufferedReader reader;
        try{
            int lineCounter = 1;
            reader = new BufferedReader(new FileReader(theFile));
            String data = reader.readLine();
            while (data != null){
                _attack.addTrainingData(data);
                data = reader.readLine();
                lineCounter++;
            }
            reader.close();
        }catch(IOException e){
            Logger.Debug("readTrainingDataFromFile has thrown a IOException", e);
        }
    }


    private static void attackFromFile(File theFile) {
        BufferedReader reader;
        try{
            reader = new BufferedReader(new FileReader(theFile));
            String data;
            Double oddsSum = new Double(0);
            Double probSum = new Double(0);
            Double nullSum = new Double(0);
            int num = 0;
            while ((data = reader.readLine()) != null) {
                if (!data.equals("")) {
                    try {
                        StatePath oddsPath = _attack.viterbiOdds(data);
                        StatePath probabilityPath = _attack.viterbiProbability(data);
                        StatePath nullPath = _attack.viterbyNull(data);
                        oddsSum += oddsPath.getScore();
                        probSum += probabilityPath.getScore();
                        nullSum += nullPath.getScore();
                        Logger.Log("Attack ("+ ++num + "):");
                        Logger.Log("Odds: "+ oddsPath.getScore());
                        Logger.Log("Probability: "+ probabilityPath.getScore());
                        Logger.Log("Null: "+ nullPath.getScore());
                        Logger.Log("");
                    } catch (ChangeVetoException e) {
                        Logger.Debug("readTrainingDataFromFile has thrown a
                            ChangeVetoException: " + data, e);
                    } catch (BioException e) {
                        Logger.Debug("readTrainingDataFromFile has thrown a BioException:
                            line " + data, e);
                    }
                }
            }
            oddsSum /= num;
            probSum /= num;
            nullSum /= num;
            Logger.Log("-----------------------");
            Logger.Log("Mean Odds = " + oddsSum.toString());
            Logger.Log("Mean Prob = " + probSum.toString());
            Logger.Log("Mean Null = " + nullSum.toString());
            Double d = new Double(oddsSum / nullSum);
            Logger.Log("Mean Odds/Null = " + d.toString());
            Logger.Log("-----------------------");

            reader.close();
        } catch(IOException e){
            Logger.Debug("readTrainingDataFromFile has thrown a IOException", e);
        }
    }

    private static String getLineFromIn(String message) {
        String line = "";
        Logger.Log(message);
        int i = 0;
        try {
            while ((i = System.in.read()) > 0) {
                Character c = (char) i;
                if (c.equals('\n') || c.equals('\r')){
                    break;
                } else {
                    line += c;
                }
            }
        } catch (IOException e) {
            Logger.Debug("Main.getLineFromIn(" + message + ")", e);
        }
        return line;
    }

}
```

Listing 5: Attack.java

```

package de.voipattack.attack;

import java.io.File;

import java.util.ArrayList;

import org.biojava.bio.BioException;
import org.biojava.bio.dist.DistributionFactory;
import org.biojava.bio.dp.BaumWelchTrainer;
import org.biojava.bio.dp.DP;
import org.biojava.bio.dp.DPFactory;
import org.biojava.bio.dp.IllegalTransitionException;
import org.biojava.bio.dp.ModelTrainer;
import org.biojava.bio.dp.ProfileHMM;
import org.biojava.bio.dp.ScoreType;
import org.biojava.bio.dp.SimpleModelTrainer;
import org.biojava.bio.dp.StatePath;
import org.biojava.bio.symbol.IllegalAlphabetException;
import org.biojava.bio.symbol.IllegalSymbolException;
import org.biojava.bio.symbol.SymbolList;

import de.voipattack.biojava.AttackSerializer;
import de.voipattack.biojava.CylcleStoppingCriteria;
import de.voipattack.biojava.PacketAlphabet;
import de.voipattack.util.Logger;

public class Attack {
  ProfileHMM _profileHMM = null;
  ProfileHMM _randomHMM = null;
  DP _dynamicProgrammingMatrix = null;
```

```java
  PacketAlphabet _alphabet = null;
  String _prefix;
  int _trainSaveModulus;

  ArrayList<String> _trainingsData = null;

  public Attack(String prefix, int trainSaveModulus) {
    Logger.Log("Attack (" + prefix + ", " + trainSaveModulus + ")");
      _trainingsData  = new ArrayList<String>();
      _prefix = prefix;
      _trainSaveModulus = trainSaveModulus;
  }

  private ProfileHMM getRandomHMM() {
    if (_randomHMM == null) {
      try {
        _randomHMM = new ProfileHMM( _alphabet.getAlphabet(), _alphabet.
            getAveragePacketNumbers(), DistributionFactory.DEFAULT, DistributionFactory.
            DEFAULT, "Random");
        ModelTrainer mt = new SimpleModelTrainer();
          //register the model to train
          mt.registerModel(_profileHMM);
          //as no other counts are being used the null weight will cause everything to be
              uniform
          mt.setNullModelWeight(1.0);
          mt.train();
      } catch (IllegalSymbolException e) {
        e.printStackTrace();
      } catch (IllegalTransitionException e) {
        e.printStackTrace();
      } catch (IllegalAlphabetException e) {
        e.printStackTrace();
      }
    }
    return _randomHMM;
  }

  public void addTrainingData(String data)  {
    if (_alphabet == null) {
      _trainingsData.add(data);
    } else {
      _alphabet.addTainingsData(data);
    }
    Logger.Log("Attack.addTrainingData(" + data + ").");
  }

  public void train(int numIter) {
    if (_profileHMM == null) {
      _alphabet = new PacketAlphabet(_trainingsData);
      _trainingsData.clear();
      // compute number of match-states as average of all trainingdata..
      Logger.Log("Attack.train() : numStates = " + _alphabet.getAveragePacketNumbers() + "
          .");

      String pHmmName = "profileHmm";
        try {
          _profileHMM = new ProfileHMM( _alphabet.getAlphabet(), _alphabet.
              getAveragePacketNumbers(), DistributionFactory.DEFAULT, DistributionFactory.
              DEFAULT, pHmmName);
        _dynamicProgrammingMatrix = DPFactory.DEFAULT.createDP(_profileHMM);
      } catch (IllegalArgumentException e) {
        Logger.Debug("Attack.train() : creating profileHMM and dynamic programming matrix"
            , e);
      } catch (BioException e) {
        Logger.Debug("Attack.train() : creating profileHMM and dynamic programming matrix"
            , e);
      }

      ModelTrainer mt = new SimpleModelTrainer();
        //register the model to train
        mt.registerModel(_profileHMM);
        //as no other counts are being used the null weight will cause everything to be
            uniform
        mt.setNullModelWeight(1.0);
        mt.train();
        Logger.Log("Attack.train() : First training complete");
    }

      //create a BW trainer for the dp matrix generated from the HMM
      BaumWelchTrainer bwt = new BaumWelchTrainer(_dynamicProgrammingMatrix);

      //anonymous implementation of the stopping criteria interface to stop after 20
          iterations
      CylcleStoppingCriteria stopper = new CylcleStoppingCriteria(_prefix,
          _trainSaveModulus, numIter,this);
      try {
        Logger.Log("Started Baum Welch Training (" + numIter + " rounds).");
      bwt.train(_alphabet.getTrainingSet(),1.0,stopper);
        Logger.Log("Finished Baum Welch Training (" + numIter + " rounds).");
    } catch (IllegalSymbolException e) {
      Logger.Debug("Debug.train(): training ", e);
    } catch (BioException e) {
      Logger.Debug("Debug.train(): training ", e);
    }
  }

  public void serialize(File file) {
    AttackSerializer.serialize(this, file);
  }

  public StatePath viterbiRandomOdds(String data) throws IllegalSymbolException,
      IllegalArgumentException, IllegalAlphabetException, IllegalTransitionException {

      DP dp = null;
    StatePath mostLikelyPath = null;
    try {
      dp = DPFactory.DEFAULT.createDP(this.getRandomHMM());
      SymbolList[] test = {_alphabet.createSymbolList(data)};
      mostLikelyPath = dp.viterbi(test, ScoreType.ODDS);
    } catch (BioException e) {
      e.printStackTrace();
    }
    return  mostLikelyPath;
  }

  public StatePath viterbiOdds(String data) throws IllegalSymbolException,
      IllegalArgumentException, IllegalAlphabetException, IllegalTransitionException {
    SymbolList[] test = {_alphabet.createSymbolList(data)};

    StatePath mostLikelyPath = _dynamicProgrammingMatrix.viterbi(test, ScoreType.ODDS);

    return  mostLikelyPath;
  }

  public StatePath viterbiProbability(String data) throws IllegalSymbolException,
      IllegalArgumentException, IllegalAlphabetException, IllegalTransitionException {
    SymbolList[] test = {_alphabet.createSymbolList(data)};

    StatePath mostLikelyPath = _dynamicProgrammingMatrix.viterbi(test, ScoreType.
        PROBABILITY);
    return  mostLikelyPath;
  }

  public StatePath viterbyNull(String data) throws IllegalSymbolException,
      IllegalArgumentException, IllegalAlphabetException, IllegalTransitionException {
    SymbolList[] test = {_alphabet.createSymbolList(data)};
```

```java

    StatePath mostLikelyPath = _dynamicProgrammingMatrix.viterbi(test, ScoreType.
        NULL_MODEL);
    return  mostLikelyPath;
  }

  public void deSerialize(File file) {
    AttackSerializer.deserialize(this, file);
  }

  public PacketAlphabet getPacketAlphabet() {
    return _alphabet;
  }

  public ProfileHMM getProfileHMM() {
    return _profileHMM;
  }

  public void setProfileHMM(ProfileHMM profileHMM) {
    _profileHMM = profileHMM;
  }

  public void setDP(DP dynamicProgrammingMatrix) {
    _dynamicProgrammingMatrix = dynamicProgrammingMatrix;
  }

  public void setAlphabet(PacketAlphabet alphabet) {
    _alphabet = alphabet;
  }
}
```

Listing 6: AttackSerializer.java

```java

package de.voipattack.biojava;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

import org.biojava.bio.BioException;
import org.biojava.bio.dp.DP;
import org.biojava.bio.dp.DPFactory;
import org.biojava.bio.dp.ProfileHMM;

import de.voipattack.attack.Attack;
import de.voipattack.util.Logger;

public class AttackSerializer {
  public static void serialize(Attack attack, File file) {
    try {
      Logger.Log("Attack.serialize(): started");
      FileOutputStream fos = new FileOutputStream(file);
      ObjectOutputStream out = new ObjectOutputStream(fos);
      out.writeObject(attack.getProfileHMM());
      out.writeObject(attack.getPacketAlphabet());
      out.close();
      Logger.Log("Attack.serialize(): Successful!");
    } catch (IOException e) {
      Logger.Debug("Attack.serialize(): Serialize the dp", e);
    }
  }

  public static void deserialize(Attack attack, File file) {
    try {
      Logger.Log("Attack.deSerialize(): started");
      FileInputStream fos = new FileInputStream(file);
      ObjectInputStream in = new ObjectInputStream(fos);
      ProfileHMM profileHMM = (ProfileHMM) in.readObject();
      DP dynamicProgrammingMatrix = DPFactory.DEFAULT.createDP(profileHMM);
      PacketAlphabet alphabet = (PacketAlphabet) in.readObject();
      in.close();
      Logger.Log("Attack.deSerialize(): Successful!");

      attack.setProfileHMM(profileHMM);
      attack.setDP(dynamicProgrammingMatrix);
      attack.setAlphabet(alphabet);
    } catch (IOException e) {
      Logger.Debug("Attack.deSerialize(): deSerialize the dp", e);
    } catch (ClassNotFoundException e) {
      Logger.Debug("Attack.deSerialize(): deSerialize the dp", e);
    } catch (IllegalArgumentException e) {
      Logger.Debug("Attack.deSerialize(): deSerialize the dp", e);
    } catch (BioException e) {
      Logger.Debug("Attack.deSerialize(): deSerialize the dp", e);
    }

  }
}
```

Listing 7: CylcleStoppingCriteria.java

```java

package de.voipattack.biojava;

import java.io.File;

import org.biojava.bio.dp.BaumWelchTrainer;
import org.biojava.bio.dp.StoppingCriteria;
import org.biojava.bio.dp.TrainingAlgorithm;

import de.voipattack.attack.Attack;
import de.voipattack.util.Logger;

public class CylcleStoppingCriteria implements StoppingCriteria{
  private int _numCycle;
  private int _saveModCount;
  private String _prefix;
  private Attack _attack;

  public CylcleStoppingCriteria(String prefix, int saveModCount, int numCycle, Attack
      attack) {
    _numCycle = numCycle;
    _prefix = prefix;
    _saveModCount = saveModCount;
    _attack = attack;
  }

  public boolean isTrainingComplete(TrainingAlgorithm ta) {
    if ((_saveModCount > 0) && (!_prefix.equals("")) && (ta.getCycle() % _saveModCount ==
        0)) {
      AttackSerializer.serialize(_attack, new File( _prefix + ta.getCycle() + ".model"));
    }
    Logger.Log("CycleStoppingCriteria : (" + ta.getCycle() + "/" + _numCycle + ")");
    return (ta.getCycle() >= _numCycle);
  }

}
```

Listing 8: PacketAlphabet.java

```java

```

```java
package de.voipattack.biojava;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import org.biojava.bio.Annotation;
import org.biojava.bio.BioException;
import org.biojava.bio.SimpleAnnotation;
import org.biojava.bio.seq.Sequence;
import org.biojava.bio.seq.db.HashSequenceDB;
import org.biojava.bio.seq.db.IllegalIDException;
import org.biojava.bio.seq.db.SequenceDB;
import org.biojava.bio.seq.impl.SimpleSequenceFactory;
import org.biojava.bio.symbol.AlphabetManager;
import org.biojava.bio.symbol.FiniteAlphabet;
import org.biojava.bio.symbol.IllegalSymbolException;
import org.biojava.bio.symbol.SimpleAlphabet;
import org.biojava.bio.symbol.SimpleSymbolList;
import org.biojava.bio.symbol.Symbol;
import org.biojava.bio.symbol.SymbolList;
import org.biojava.utils.ChangeVetoException;

import de.voipattack.util.Logger;

public class PacketAlphabet implements Serializable{
  private static final long serialVersionUID = 8804321526083471878L;

  private FiniteAlphabet _alpha = null;
  SequenceDB _trainingSet = null;
  private int _averagePacketNumber = 0;
  HashMap<String, Symbol> _symbols = new HashMap<String, Symbol>();

  @SuppressWarnings("unchecked")
  public PacketAlphabet(ArrayList<String>traindata) {
    ArrayList<ArrayList<Symbol> > symbollist = new ArrayList<ArrayList<Symbol>>();
    int sum = 0;

    for (String data : traindata) {
      String packets[] = data.split(" ");

      sum += packets.length;
      ArrayList<Symbol> sList = new ArrayList<Symbol>();
      for (String packet : packets) {
        String size = "";
        if (packet.contains(",")) {
          size = packet.split(",")[1];
        } else {
          size = packet;
        }
        if (!size.equals("")) {
          Integer sizei = Integer.parseInt(size);
          Symbol s = new PacketSymbol(sizei, Annotation.EMPTY_ANNOTATION);
          if (_symbols.containsKey(sizei.toString())) {
            s = _symbols.get(sizei.toString());
        }  else {
            _symbols.put(sizei.toString(), s);
          }
          // assuming hash set does not have any double entries.
          sList.add(s);
        }
      }
      symbollist.add(sList);
    }

    _averagePacketNumber = sum / traindata.size();
    Set symbolSet = new HashSet<Symbol>();
    for (Symbol s : _symbols.values()) {
      symbolSet.add(s);
    }
    _alpha = new SimpleAlphabet(symbolSet, "PacketAlphabet");

      //iterate through the symbols to show everything works
      String log = "Alphabet created : ( ";
      for (Iterator i = _alpha.iterator(); i.hasNext(); ) {
        Symbol sym = (Symbol)i.next();
        if (i.hasNext())
          log += sym.getName() + ", ";
        else
          log += sym.getName();
      }
      Logger.Log(log + ").");

    AlphabetManager.registerAlphabet("PacketAlphabet", _alpha);
    _trainingSet = new HashSequenceDB();
    SimpleSequenceFactory factory = new SimpleSequenceFactory();

    for (ArrayList<Symbol> sList : symbollist) {
      try {
        SimpleSymbolList ssl = new SimpleSymbolList(_alpha, sList);
        Sequence s = factory.createSequence( ssl, "", ssl.seqString(), new
            SimpleAnnotation());
        _trainingSet.addSequence(s);
      } catch (IllegalSymbolException e) {
        Logger.Debug("PacketAlphabet", e);
      } catch (IllegalIDException e) {
        Logger.Debug("PacketAlphabet", e);
      } catch (ChangeVetoException e) {
        Logger.Debug("PacketAlphabet", e);
      } catch (BioException e) {
        Logger.Debug("PacketAlphabet", e);
      }
    }
  }

  public SequenceDB getTrainingSet() {
    return _trainingSet;
  }

  public FiniteAlphabet getAlphabet() {
    return _alpha;
  }

  public int getAveragePacketNumbers() {
    return _averagePacketNumber;
  }

  public SymbolList createSymbolList(String data) throws IllegalSymbolException {
    String packets[] = data.split(" ");

    ArrayList<Symbol> sList = new ArrayList<Symbol>();
    for (String packet : packets) {
      String size = "";
      if (packet.contains(",")) {
        size = packet.split(",")[1];
      } else {
        size = packet;
      }
      Integer sizei = Integer.parseInt(size);
      if (_symbols.containsKey(sizei.toString()))
        sList.add( _symbols.get(sizei.toString()));
      else
        throw new IllegalSymbolException("The symbol '" + size + "' was not in the
            Alphabet!");
    }
    return new SimpleSymbolList(_alpha, sList);
  }

  public void addTainingsData(String data) {
    try {
      SymbolList l = createSymbolList(data);
      Sequence s = new SimpleSequenceFactory().createSequence( l, "", data, new
          SimpleAnnotation());
      _trainingSet.addSequence(s);
    } catch (IllegalSymbolException e) {
      Logger.Debug("PacketAlphabet.addTrainingsdata(" + data + ").", e);
    } catch (IllegalIDException e) {
      Logger.Debug("PacketAlphabet.addTrainingsdata(" + data + ").", e);
    } catch (ChangeVetoException e) {
      Logger.Debug("PacketAlphabet.addTrainingsdata(" + data + ").", e);
    } catch (BioException e) {
      Logger.Debug("PacketAlphabet.addTrainingsdata(" + data + ").", e);
    }
  }
}
```

Listing 9: PacketSymbol.java

```

package de.voipattack.biojava;

import org.biojava.bio.Annotation;
import org.biojava.bio.symbol.FundamentalAtomicSymbol;


public class PacketSymbol extends FundamentalAtomicSymbol {
  private static final long serialVersionUID = -42925484312330097L;

  public PacketSymbol(Integer num, Annotation annotation) {
    super(num.toString(), annotation);
  }
  @Override
  public boolean equals(Object obj) {
    return (obj instanceof PacketSymbol) ? this.getName().equals(((PacketSymbol) obj).
        getName()) : false;
  }

  @Override
  public int hashCode() {
    return getName().hashCode();
  }
}
```

Listing 10: Logger.java

```

package de.voipattack.util;

import java.io.PrintStream;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Logger {
  public enum LogLevel {LOW, MIDDLE, HIGH};

  private static PrintStream  _errStream;
  private static PrintStream _outStream;
  private static SimpleDateFormat _format;

  static {
    _errStream = System.err;
    _outStream = System.out;
    _format = new SimpleDateFormat("hh:mm:ss");
  }

  public static void Debug(Exception e) {
    Logger.Debug("", e);
  }

  public static void Debug(String message, Exception e) {
    Logger.Debug(message, LogLevel.HIGH, e);
  }

  public static void Debug(String message, LogLevel dl, Exception e) {
    if (_errStream != null) {
      _errStream.println("Debug (" + _format.format(new Date()) + "): " + message);
      _errStream.println(e.getMessage());
      _errStream.println();
    }
  }

  public static void Log(String message) {
    Logger.Log(message, LogLevel.LOW);
  }

  public static void Log(String message, LogLevel dl) {
    if (_outStream != null)
      _outStream.println("Log (" + _format.format(new Date()) + "): " + message);
  }

  public static void setOutStrem(PrintStream stream) {
    _outStream = stream;
  }

  public static void setErrStream(PrintStream stream) {
    _errStream = stream;
  }
}
```