Probabilistic and Genetic Attacks on the Key-Exchange Protocol Using Permutation Parity Machines.

Luís F. Seoane
Bernstein Center for Computational Neurosciences, Technische Universität Berlin.

Supervisor: Dr. Andreas Ruttor
Group Methods of Artificial Intelligence, Technische Universität Berlin

Three strategies are presented to attack the key-exchange protocol based on Permutation Parity Machines (PPM). Two of the strategies are based on probabilistic considerations: one applies only analytical procedures and the other one implements Monte Carlo methods to attack the protocol. The third strategy consists on a Genetic Algorithm. The performance of the three strategies is analyzed for PPM with $K = 1$, yielding promising results for the non-analytic procedures and fine results for the analytic one. A combination of the analytic and the Monte Carlo procedures is employed for PPM with $K = 2$. Its performance is also analyzed, which again yields very promising results.

PACS numbers: 84.35.+i, 87.18.Sn, 89.70-a, 05.10.Ln

INTRODUCTION

The study of interacting neural networks [1, 2] led to the discovery of synchronization between two parties. This synchronization showed to be fast and secure enough as to develop the field of *neural cryptography* [3–9] during the last decade. The idea is that two neural networks A and B–originally, multilayer perceptrons with independent receptive fields named Tree Parity Machines (TPM)–with initial private weights compute their output states τ_A and τ_B when an example (encoded in input vectors $\overrightarrow{x}$) is presented. If the networks agree on their outcomes ($\tau_A = \tau_B$), they apply some learning algorithm (e.g. Hebbian or anti-Hebbian learning). Repeating this process with several input vectors would eventually conduct to full synchronization of the two networks (i.e. the weights in both networks would be the same, though they remained private the whole time).

This synchronization was shown to be fast: meaning that a third party E would need more examples to successfully synchronize with the other two networks, despite the many strategies tested [3–11]. Nowadays, *neural cryptography* is a promising field for key-exchange protocols and finding any weakness in a neural system capable as a cryptographic means would be exciting for the advance of the field in either direction (reassuring the security of *neural cryptography* or showing flaws to be prevented).

A few *classic attacks* (*classic*, as opposed to the probabilistic attacks introduced in this work, but which efficiency has been suggested in [5, 9]) were tried against TPM [3–5, 7, 8, 10, 11]. These attacks involved one third TPM or assemblies of them, and they performed wrong enough as to consider that the TPM key-exchange protocol is safe.

This paper is centered in the study of new attacks against Permutation Parity Machines (PPM) [9]. These new machines (slightly different from TPM) showed a greater robustness than TPM against classic attacks. Thus, any new advance would have a greater impact if it were tested on PPM than if it were tried on TPM, and the impact would be even greater if the methods developed were also suitable for TPM.

The attacks proposed here are probabilistic attacks based on an attempt to guess the *weights* of the parties by calculating some posterior probabilities after the examples $\overrightarrow{x}$ and outcomes τ_A and τ_B are known. Also a Genetic Algorithm is proposed, which differs form the previous genetic strategies used in TPM.

The neural networks studied in this work have two layers with K hidden units in the first layer. The number of hidden units in this layer plays a major role, since only systems with $K > 1$ are valid for *neural cryptography*. In a network with $K = 1$, the state τ of the network–which is made public–is the same as the state of the only hidden layer. But for the system to have cryptographic utility, the state of the hidden units must be kept hidden and this only happens for $K > 2$.

In this work it is first studied the performance of the new attacks for $K = 1$. Although this case would lack interest for *neural cryptography*, it works as a toy model to test the attacks. If the new ideas were to have any interest, they must at least perform very well in the case $K = 1$.

Apart from this, $K = 1$ has also a *recreative* motivation. The information made available by a PPM in this case is related to the number of bits from the input vector which match the bits from the weights (description below). This is similar to the game *mastermind*, although the information made public in the present case is less than in the game: it is only said if the number of *right guesses* is above a threshold. Also, the alphabet is restricted to a binary one: the weights and the inputs to the PPM are only 0 or 1 as explained in the next section, while in *mastermind* usually an alphabet of many colors is used. Thus, solving this problem would have some implications for the game as well.

In a later part of the work, the attacks which performed fine enough for $K = 1$ were also tested for $K = 2$ as it is reported in the present paper.

DESCRIPTION OF THE MACHINE AND THE ATTACKS

Let N, K and $G >> N \cdot K$ be positive integers. A PPM is a neural network with two layers: K hidden units in the first layer, each of which is a perceptron with an independent receptive field of size N; and one only unit in the second layer. The j-th hidden unit has an internal state $\sigma_j \in \{0,1\}$, and the PPM itself has an outcome $\tau = \oplus_{j=1}^{K} \sigma_j$ (i.e. the parity of the states of the hidden units), which is also the state of the unit in the second layer.

For the computation of the state of the j-th hidden unit, an input vector $\overrightarrow{x_j}$ is provided. The weights $\overrightarrow{w_j}$ of the j-th unit are drawn from a pool of random numbers common to all the units: the state vector $\overrightarrow{s}$ of size G. The input vector and the weights are both random bits with value $x_{i,j} \in \{0,1\}$, $w_{i,j} \in \{0,1\}$. The vector local field $\overrightarrow{h_j}$ of the hidden unit is defined as the one-by-one logical XOR operation of the bits in $\overrightarrow{x_j}$ and $\overrightarrow{w_j}$: $\overrightarrow{h_j} = \overrightarrow{x_j} \oplus \overrightarrow{w_j} \Rightarrow h_{i,j} = x_{i,j} \oplus w_{i,j}$; and the scalar local field h_j is defined as the number of 1s in $\overrightarrow{h_j}$: $h_j = ||\{h_{i,j} | h_{i,j} = 1; i = 1, ..., N\}||$. The state of the j-th unit is $\sigma_j = \theta(h_j - N/2)$, where θ is the Heaviside step function with $\theta(0) = 0$.

As pointed out before, in the state vector $\overrightarrow{s}$ we find random bits $s_i \in \{0,1\}$, appearing both values with equal pobability. To draw the weights of the units from the pool of random bits $\overrightarrow{s}$, the PPM is assisted by a matrix π of size $K \cdot N$. The entries of the matrix π are such that $1 \leq \pi_{i,j} \leq G, \forall i,j \in \{1, ..., N\}x\{1, ..., K\}$. Provided such a matrix, the weights can be set up: $w_{i,j} = s_{\pi_{i,j}}$.

When performing synchronization tasks, two PPM (A and B) designed with the same settings (i.e. with same N, K and G) are provided. The synchronization will succeed after several *inner* and *outer rounds*, which are described below. For each *inner round*, a matrix π and input vectors $\overrightarrow{x_j}$ for each hidden unit are also provided. The entries of the matrix π are drawn homogeneously and by random from the set $\{1, ..., G\}$. In each *inner round*, both PPM compute their outputs τ_A and τ_B and if they agree ($\tau_A = \tau_B$), they store the state of their first hidden units in a buffer which remains private for each PPM.

Thus, there are: public and common input vectors $\overrightarrow{x_j}$ and π matrix; public but not necessarily equal outcomes τ_A and τ_B; private, not necessarily equal state vectors $\overrightarrow{s_A}$ and $\overrightarrow{s_B}$; and private, not necessarily equal states of the first hidden units σ_1^A and σ_1^B.

The *inner rounds*, are repeated until the buffers where σ_1^A and σ_1^B are stored have size G. Then, a so-called *outer round* is completed and the buffer with the successive states of the first hidden units replace the corresponding state vectors in each machine. The dynamics of the PPM are such that after each *outer round* the state vectors $\overrightarrow{s_A}$, $\overrightarrow{s_B}$ tend to be more alike, eventually reaching full synchronization $\overrightarrow{s_A} = \overrightarrow{s_B}$. The synchronization time (in number of *outer rounds*) grows in a polynomial fashion with the size of the input (N) and the size of the state vector $\overrightarrow{s}$ (G).

Reported previous attacks on PPM [9] showed very poor performance of single PPM and of ensembles of such machines which try to mimic the behavior of the synchronizing ones. Namely, the time –in outer rounds– after which the Eavesdroppers synchronize grows exponentially.

In the following, the description of three new strategies are presented. These ones do not pursue to mimic the synchronizing process, but to first guess the state vector of A (or B) during an *outer round* and consecutively to reproduce the state vector in the next *outer round*.

Probabilistic attacks

For sake of simplicity, some notation is introduced now. The synchronizing parties A and B will be eavesdropped by a third agent E not necessarily implementing a PPM itself. If it exists, the state vector of E's PPM will be called $\overrightarrow{s_E}$ and the outcome of its machine will be called τ_E. Rather than in this objects, the calculi performed by E will usually rely on some probabilistic entities. Namely, the probability that the i-th bit of $\overrightarrow{s_A}$ is 0 (based on E's knowledge of $\overrightarrow{s_A}$) is called p_i, and the array with the probabilities that each bit from $\overrightarrow{s_A}$ is 0 is called $\overrightarrow{p_E}$ Usually, provided these probabilities $\overrightarrow{s_E}$ will be identified as the most probable state vector.

To measure the level of synchronization between two state vectors it is introduced the *normalized overlap* $\rho^{AB} = (G - d_H(\overrightarrow{s_A}, \overrightarrow{s_B}))/G$, where $d_H(\overrightarrow{s_A}, \overrightarrow{s_B})$ is defined as the Hamming distance of the arrays $\overrightarrow{s_A}$ and $\overrightarrow{s_B}$. The same definition holds for the normalized overlap of any state vector with $\overrightarrow{s_E}$. It is also possible to define an averaged overlap between $\overrightarrow{s_A}$ or $\overrightarrow{s_B}$ and $\overrightarrow{p_E}$ as: $\langle \rho^{AE} \rangle = \sum_{i=1}^{G} [(1 - s_{A_i}) \cdot p_{E_i} + s_{A_i} \cdot (1 - p_{E_i})]$.

Usually, it will be necessary to transfer the knowledge of $\overrightarrow{s_A}$ that E has achieved during an *outer round* into the next one. In these cases, we will identify $\overrightarrow{p_E}$ and p_i as the known probabilities in the previous *outer round* and $\overrightarrow{p_E}^+$ and p_i^+ as the probabilities of each bit being either 0 or 1 in the next *outer round*. If the old *outer round* is not referred anymore in the text, both $\overrightarrow{p_E}^+$ and p_i^+ will be noted as $\overrightarrow{p_E}$ and p_i in the following.

In a probabilistic attack a prior probability is supposed, that each bit in the state vector is either 0 or 1. During this report it is taken: $p_i = p(s_i = 0) \Rightarrow p(s_i = 1) = 1 - p_i$. It is understood that we try to guess the state vector of the machine A. Since no previous information about $\overrightarrow{s_A}$ is available to E, the usual hypothesis is $p_i = 0.5 \ \forall i = 1, ..., G$.

In each *inner round* it is provided an input $\overrightarrow{x} = \{\overrightarrow{x_j}, \ j = 1, ..., K\}$ and it is imposed that the outcome should be τ_A. The posterior probability $p(s_i = 0 | p_i, \overrightarrow{x}, \tau_A)$ is calculated by some means, and the probabilities of each bit being either 0 or 1 are updated $p_i \longrightarrow \tilde{p} = p(s_i = 0 | p_i, \overrightarrow{x}, \tau_A)$. This process is repeated for the next *inner round* with the new p_i.

The method to calculate the posterior probability should make the p_i converge towards 0 (meaning $p(s_i = 0) = 0 \Rightarrow s_i = 1$) or 1 ($p(s_i = 0) = 1 \Rightarrow s_i = 0$) after several *inner rounds*. If the algorithm succeeds, $s_i = s_{A,i}\ \forall i = 1, ..., G$ after all the p_i have collapsed. If one or more p_i have collapsed to the wrong value, a case might be given in which the desired output can not be generated. This means that the estimation of the p_i was wrong and a mechanism might be provided to restore the initial no-info situation for some of the bits involved (this means that some of the bits might return to $p_i = 0.5$). This prevents a numerical implementation of a probabilistic attack from getting stuck in an infinite loop.

In the **Mean Field** algorithm purely analytical steps are taken to calculate the posterior probability. Here the algorithm is described for $K = 1$, but a generalization to larger K is straightforward if some hypothesis about the independency of the weights are made. For example, for $K > 1$ it is convenient to suppose that each bit from the state vector appears maximal once in each *inner round*. For the sake of simplicity this hypothesis is also made for $K = 1$. This means: each matrix π has not got twice the same entry. This assumption is right in the limit $G >> N \cdot K$.

For the analytical development, we also suppose that all the bits in the input vectors are 0. This transforms the problem of calculating the local field of one unit into scoring the number of 1s in the weights. There is no loss of generality here because if the bits in $\overrightarrow{x}$ were allowed to be 1, we would only have to take the complementary probability $\hat{p}_i = 1 - p_i$ for the calculi in the procedure described below whenever this situation happened.

Thus, provided the prior p_i for all the bits in the weights (now $i = 1, ..., N$), the posterior for each bit is calculated in a manned inspired by the Gibbs's sampling process [12]. Consecutively, each bit i is left out and the probability distribution of the local field (i.e. $p(h = n|n = 0, ..., N-1)$ –and it always holds $h \neq N$ since there are only $N - 1$ bits left) is calculated taken into account only the $N - 1$ remaining bits $p_{j\neq i}$. Eventually, the posterior probability of the i-th bit is the proportion of cases in which this bit can be zero and the conditions ($p(h = n|n = 0, ..., N)$, $\overrightarrow{x} = \overrightarrow{0}$ and $\tau = \tau_A$) are obeyed.

To calculate the probability distribution of the local field a mean field point of view is taken (thus the name of the algorithm). The number of ones in $\overrightarrow{w}$ when the i-th bit is left out is estimated by a binomial distribution with parameter $p^j_\mu \equiv \sum_{j\neq i}(1 - p_j)/(N - 1)$. This approximation is exact if and only if $p_{j\neq i}$ is the same for all j. In this picture, the remaining bits constrain the i-th one in an averaged fashion which reminds of mean field interactions.

In the **Monte Carlo** attack the posterior probability is not analytically calculated, but it is estimated from a sampling of all the possible state vectors compatible with the desired conditions ($\overrightarrow{x}$ and τ_A). It has the inconvenience that the results are mere approximations to the actual probability distribution and that the sampling can take longer than the analytic calculi, but it does not need any assumptions on the correlation of the weights: it is not needed to suppose that each bit from $\overrightarrow{s}$ shows only once in each π matrix.

For each *inner round*, and provided the p_i of the bits involved (i.e. of the weights); random samples are generated. These are 0s or 1s drawn by random according to the corresponding p_i. These bits randomly generated sample the space of possible weights. For each such a sample, the outcome of the machine is calculated. If $\tau = \tau_A$, the sample is stored as valid; otherwise it is neglected. After a large enough sample has been generated, the proportion of cases in which the i-th bit is zero out of the total number of valid cases becomes the posterior probability p_i.

In the case $K = 1$ it does not make any sense to study the performance after several *outer rounds* because the state σ_1 of the only hidden unit is also the outcome τ of the PPM after each *inner round*, and this is public. This means that PPM with $K = 1$ can not work for neural cryptography. However, if it is desired to research a case with $K > 1$, it should eventually be needed to investigate the performance of the algorithms after more than one *outer round*. To do so, it must be provided a mechanism to transfer the previous knowledge about $\overrightarrow{s}$ into the new state vector $\overrightarrow{s}^+$.

Given the $\overrightarrow{p}_E$ after the chosen algorithm has been applied on a whole *outer round*, the idea is to calculate the maximum-a-posteriori probability that the first hidden unit has a certain internal state provided an input $\overrightarrow{x}$, a matrix π and an output τ_A, which is all the public information we can use. It was tried to generate $\overrightarrow{p}^+_E$ through Monte Carlo simulations and with a version of the Mean Field analytic method which included all the bits to estimate the parameter of the binomial (in opposition to the leave-one-out method). The Monte Carlo procedure was in this issue particularly prone to get stuck, provoking the reset of many bits –as mentioned above, a mechanism was provided to reset some p_i when the estimation proved to be wrong– and leading to the consecutive loss of information. The procedure inspired in the Mean Field approach to calculate the maximum-a-posteriory $\overrightarrow{p}^+_E$ was better for this task and was eventually used to transfer information from an *outer round* into the next one whenever needed.

The procedure is similar to the one described before. Now, instead of leaving one bit out, the probability distribution of the local field $p(h = n|n = 0, ..., N)$ is calculated taking all the bits into account by using the average $p_\mu \equiv \sum_i (1-p_i)/N$ as the parameter of a binomial describing the number of zeros in $\overrightarrow{w}$. From here, it can be calculated the probability that each hidden unit has a certain state ($p(\sigma_j = 0) = \sum_{n=1}^{N/2} p(h = n)$) and eventually the probability that the machine has a certain output (for $K = 2$, $p(\tau = 0) = p(\sigma_1 = 0) \cdot p(\sigma_2 = 0) + p(\sigma_1 = 1) \cdot p(\sigma_2 = 1)$).

Genetic attack

In a **Genetic Algorithm** a population of test solutions is considered, which seek for the optimization of some *goal function*. The performance of each test solution is evaluated by means of a *fitness function* which should somehow resemble the *goal function*. The best test solutions are selected, combined with each other after some *mixing* rules and mutated to prevent the algorithm from getting stuck in a local extremum.

In the present case, two PPM A and B are given which perform a synchronizing task. The set of test solutions $T = \{\overrightarrow{s_1}, ..., \overrightarrow{s_{n_T}}\}$ (of size n_T) is composed of possible state vectors generated by random. The problem is to maximize the overlap between the test solutions and s_A (this overlap is the *goal function*); but s_A is not available to anyone but A, so we can not calculate the *goal function*. Since the PPM A and B do not change their state vectors during a whole *outer round*, a suitable and available *fitness function* for the i-th test solution $\overrightarrow{s_i}$ is the number of cases in which a PPM E with $\overrightarrow{s_i}$ as state vector has the same outcome as the machine A ($\tau_E = \tau_A$) during a single *outer round*. In the implementation of the algorithm, this number is evaluated for each test solution in T. Also for each single bit in each $\overrightarrow{s_i}$ the number of right guesses ($\tau_E = \tau_A$) in which the bit is involved is tracked.

The $n_T/2$ fittest test solutions (those which yield a larger amount of matches with the PPM A) are selected and mixed. For mixing, the test solutions are taken pairwise and one by one the performances of each individual bit are compared. The bits performing better become part of a new test function and those performing worse become part of another new test function. It is expected that a new, fitter test function is generated together with a new, not-so-fit test function.

Genetic Algorithms are prone to get stuck in local extrema of the *fitness function*. The *mutation* of some of the elements– i.e. to flip some bits in each test solution with some probability p_{mut}–adds variety to the test solutions and prevents up to some degree this flaw. The top performing test solutions were protected from this mutation step to preserve their good performance.

The Genetic Algorithm should increase the average overlap of the fittest test solutions after one *outer round*, but this procedure can be repeated using the same *outer round* as many times as possible. Usually, this is done until no improvement of the *fitness function* is noticed, which would mean that the algorithm can not extract more information from the current *outer round*; or until a number of runs over the same *outer round* is completed. After one of these *stop conditions* is reached, it is time to generate the state vector of the next *outer round* for each of the test functions in T. After this is done, the algorithm might go on using the next *outer round*. In this report it is analyzed the performance of the Genetic Algorithm in PPM with $K = 1$. This, as explained before, makes pointless the research beyond the first *outer round*.

RESULTS

It is reported the performance of the three algorithms for machines which $K = 1$, beginning with the Mean Field attack, following with the Monte Carlo attack and ending with the Genetic Algorithm. The results are fine for the first approach and very promising for the other two. However, the nature of the PPM with $K = 1$ makes them unsuitable for neural cryptography. Therefore, the problem with $K = 1$ is not so interesting as with larger K. A nice application of the algorithms would be that of playing *mastermind* with a computer, which is a similar problem to the one presented here.

The problem becomes much more complicated for $K > 2$. Some symmetries of the PPM make it quite difficult to guess the state vectors of these machines and most of the ideas tried did not work. These machines are actually suitable for neural cryptography, so breaking the protocol for these machines would be the interesting problem. The only algorithm which performed good enough for $K = 2$ is the Mean Field approach to calculate the maximum-a-posteriory probabilities p_i^+ of a following *outer round* based on the p_i of the previous one, combined with a Monte Carlo sampling to guess the p_i within each *outer round*. This result is reported at the end of this section.

Performance on PPM with $K = 1$

The **Mean Field** algorithm to guess the state vector of a PPM is the one which performed worst. We can identify the main source for this low performance in the assumptions we made. There exist some correlations between the different variables that we might be neglecting (e.g. a same bit might be present twice in the weight of a unit). Also, once the assumptions have been made, the calculi are purely analytical and the algorithm should suffer some information loss due to the degeneracy of the possible keywords that a PPM can generate. This degeneracy in the number of keywords could be accentuated by the neglected correlations.

In figure 1 it is presented the performance of the algorithm during one *outer round*. The results are averaged over 100 simulations and both the average overlap $\langle \rho^{AE} \rangle$ (corresponding to the overlap of p_E with s_A) and the overlap of the most probable solution ρ^{AE} are presented. We can appreciate how the average overlap is increased above a 56% and the overlap of the most probable solution is increased above a 60%. Notwithstanding the increase of the overlap, we can not guess the state vector completely. Thus, the performance of the algorithm is just fine.

The **Monte Carlo** attack on PPM showed an outstanding performance to guess the state vector of a PPM. Let us note that, for PPM with a given N, in each *outer round* and without any knowledge of the state vector there are 2^N possible weights that each hidden unit can have, and around $2^N/2$ are compatible with either outcome of the PPM τ_A, so the size of

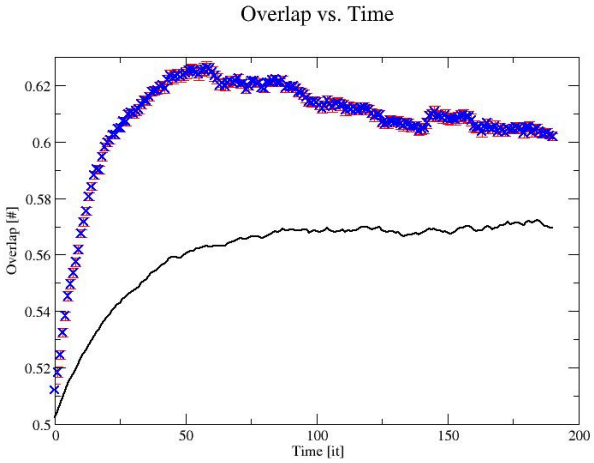

FIG. 1. **Performance of the Mean Field attack: Overlap vs. Time.** Evolution of the overlap with time during one single *outer round* when applying a Mean Field algorithm to guess a state vector. The results are averaged over 100 simulations. The standard deviation is used to draw the error bars. Since different simulations could have different number of *inner rounds*, the data are presented only up to the length of the shortest *outer round*. Black line: average overlap $\langle \rho^{AE} \rangle$. Blue crosses (with red error bars): overlap of the most probable state vector ρ^{AE}. The time is measured in *inner rounds*. The design settings of the PPM are: $N = 8$, $G = 128$ and $K = 1$.

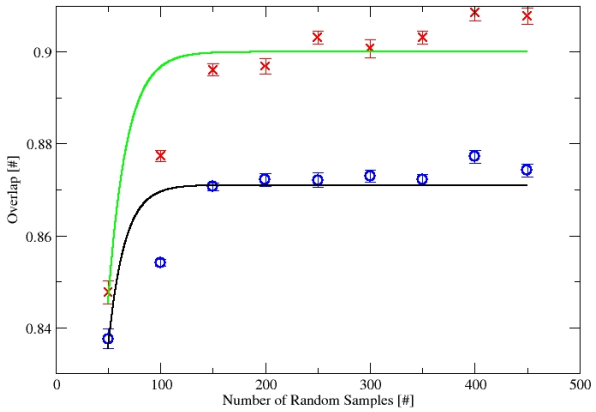

FIG. 2. **Performance of the Monte Carlo attack: overlap vs. size of sampled space.** Blue circles: average overlap $\langle \rho^{AE} \rangle$. Red crosses: overlap of the most probable state vector ρ^{AE}. Black and green: fits of the data to $f(x) = A(1 - e^{-Bx})$. The fits are only included as a qualitative hint of the tendency. Results are averaged over 10 simulations. The standard deviation is used to draw the error bars. The settings of the macchines are $K = 1$, $G = 128$ and $N = 8$. The size of the sampled space is $2^8 = 256$, so the sampling spans from less than a 20% to around a 175% showing good performance for any of the values.

the space with the possible state vectors is huge and grows exponentially with N. When using the Monte Carlo algorithm, we sample this space generating only a portion of the state vectors which are compatible with each desired outcome. This sampling implies that the estimation of the p_i is not as accurate as that from the Mean Field approach, but this also avoids some effects of the degeneracy of the system. The algorithm is prone to get stuck, which means that extremely wrong estimations of the p_i are often done. To dodge this problem, when it is hard to generate valid weights compatible with a desired output τ_A (i.e. a lot of random weights are generated according to the p_i and none of them matches τ_A), one of the p_i is supposed to be wrong and reset to $p_i = 0.5$. The performance of the algorithm shows that this combination works both to avoid wrong estimation of the p_i in a long term and to overcome the limits that the degeneracy of the keywords impose to the probabilistic methods.

In figure 2 it is presented the performance of the method when varying the size of the sampling (from less than 20% to arund 175%). We see that in all the cases the method manages to increase the average overlap above 70%, as well as the overlap of the most probable solution. For large samplings, the average overlap $\langle \rho^{AE} \rangle$ raises above 87%.

The **Genetic Attack** on PPM also showed an outstanding performance. In figure 3 it is presented the performance of the algorithm when varying the number of times that the algorithm is applied over the same *outer round*. In the Genetic attack a population of possible state vectors is considered (the test solutions), but only the one which performs better is taken into account to calculate the performance of the method. So, we see how the best performing test solution reaches an overlap just below 80% if only 10 runs over the same *outer round are considered*. This overlap can be held above 86% for 30 runs over the same *outer round*.

A last comment on the performance for $K = 1$ which might be of great interest for further application of these attacks is one related to the information that the algorithms get from one single *outer round*. As in the case of the Genetic attack, where the same *outer round* is used several times to evaluate and select those test solutions matching more of their outcomes τ_E with τ_A; the methods described here can use many times the same *outer round* until all the possible information has been extracted.

We could expect that –again, ignoring that some correlations are neglected– the Mean Field probabilistic approach would actually be capable of extracting all the information from one *outer round* since all the steps taken are analytical. When this is the case, the performance of the method is hindered by the degeneracy in the number of keywords that the PPM can generate. In figure 4 we can see how the entropy of the solution generated by the Mean Field attack decreases monotonously, but it is always kept in a value greater than 0.

For the Monte Carlo, and based on some toy simulations, it seems possible to extract more information out of each *outer*

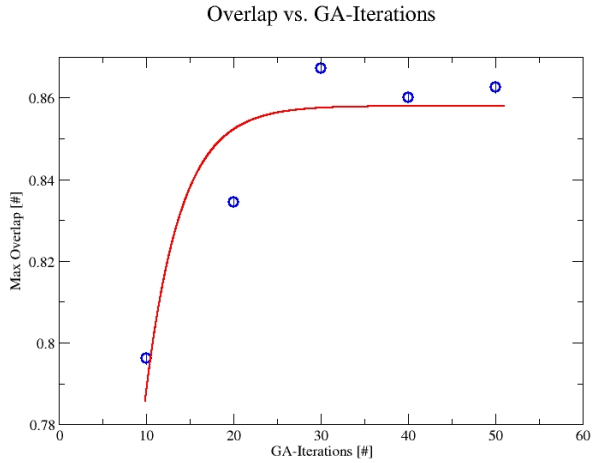

FIG. 3. **Performance of the Genetic Attack: overlap vs. number of runs over the same *outer round*.** Blue circles: overlap averaged over 10 simulations. Standard deviation is used for the error bars. Red: fit of the data to $f(x) = A(1 - e^{-Bx})$. Again, it is only presented for qualitative comprehension of the tendency. The settings of the PPM are: $N = 8$, $K = 1$, $s = 128$.

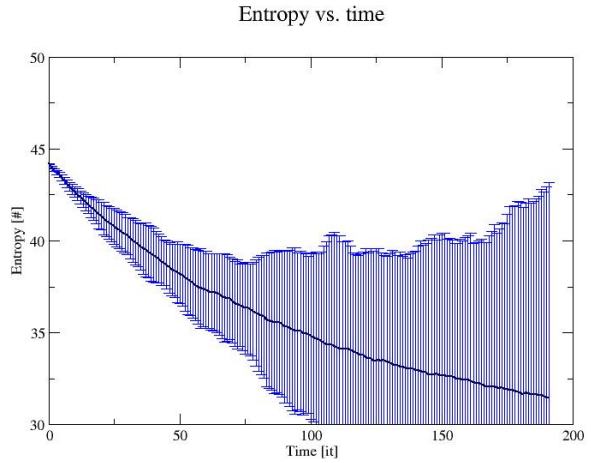

FIG. 4. **Performance of the Mean Field attack: entropy of the solution.** Black: entropy of p_E averaged over 100 simulations with the standard deviation depicting the corresponding error bars. The settings of the PPM are $N = 8$, $K = 1$ and $G = 128$. A huge entropy points out that the number of keywords compatible with the successive examples (input and τ_A) hinders the convergence of the analytic method.

round, as seen in figure 5. Here, with the same settings as before, one single simulation is run many times over the same *outer round*. Each run has around 250 *inner rounds*, meaning that the peak of overlap is reached at 1.5 *outer rounds* more or less. After this, the method is not able to extract more information and shows a decay in the performance which could be also associated to the degeneracy of the keywords, as it happens in the case of the Mean Field approach.

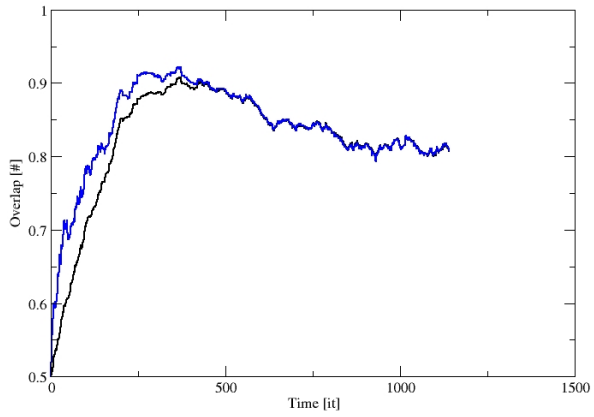

FIG. 5. **Performance of the Monte Carlo attack: using many times the same *outer round*.** Black: averaged overlap. Blue: overlap of the most probable state vector. The time is measured in *inner rounds* and only one simulation is presented. The peak happens at about 1.5 *outer rounds*, having each *outer round* around 250 *inner rounds* in this case. After the performance peaks, a decay is observed which is coherent with the limitations imposed by the degeneracy of the keywords. The settings of the PPM are $N = 8$, $K = 1$ and $G = 128$.

Performance on PPM with $K = 2$

Each of the methods presented serious problems when considering synchronizing PPM with $K = 2$. Of course, this is the most interesting case (in general, $K > 1$) because now the PPM could work for cryptographic aims.

The Mean Field analytic method did not show a performance good enough for $K = 1$, and because of time constrains it was left out of the experiments with $K = 2$.

Many simulations were done with the Genetic Attack, also trying different combinations of the number of test solutions and the number of runs over the same *outer round*. However, this method did not show a good performance in any of the cases. The overlap of the best test solution was stable oscillating around 62% and did not show any improvement with the different attempts, so this method lose the interest that it showed for $K = 1$.

When analyzing the performance of the **Monte Carlo** guessing for $K = 2$ during one single *outer round*, the method showed a monotonous increase of the averaged overlap, peaking above 60% or 80% depending on the settings of the PPM. The overlap reached was not the most important part (the Genetic Attack also reached a 60% for some settings), but the fact that the grow of the overlap was sustained –while the genetic algorithm reached a maximum when beginning its appli-

cation and could not improve further–. This pointed out that the method was continuously obtaining information from the synchronizing process, which the Genetic Algorithm failed to do; and hinted that further information could be obtained if the knowledge about p_i were successfully conveyed to p_i^+.

The problem of this algorithm arose when attempting to generate the maximum-a-posteriory p_i^+ for a consecutive *outer round*. When doing so, the chances that the method takes (by sampling only a small portion among all the possibilities) were too important and the generated p_i^+ were prone to get stuck, meaning that more and more bits would be reset to $p_i = 0.5$. Since no guessing is required when calculating p_i^+ out of p_i, an analytical estimation of p_i^+ is preferred. Therefore, the maximum-a-posteriori p_i^+ were calculated based on the Mean Field approach.

The settings of the studied PPM are $N = 8$, $K = 2$ and $G = 128$, as the machines used in [9]. The number of possible weights in each inner round is $2^{2\cdot 8} = 2^{16} = 65536$ but only $65536/2 = 32768$ of these weights are compatible with the desired outcome τ_A. The size of the sampling was 1000, which means that only a 3% of the actual possibilities were considered.

In figure 6 it is represented the evolution of the averaged overlap over time (black) and the overlap of the most probable state vector. The first and very important result is that both of them converge to 1, meaning that the method is efficiently capable of guessing the internal state of the PPM in a finite time. Even more, after 3 *outer rounds* (time spanned by the simulations) the synchronizing PPM had not been able to synchronize yet. This means that, for this settings of the PPM ($N = 8$, $K = 2$ and $G = 128$) the Monte Carlo algorithm combined with a Mean Field method to transfer the knowledge about p_i from an *outer round* to the next one is faster than the synchronization of two PPM.

A curious phenomenon to be noted in the plot is the decrease of the overlap happening between the *inner rounds* 500 and 600 more or less. At this time should also be taking place the transitions from the second *outer rounds* to the thirds ones (let us note again that each of the pair of PPM considered in each simulation transited from the second to the third *outer rounds* at different times), and the lose of information might be related to it. However, this phenomenon did not show up in the first transition between *outer rounds*.

To further support this good result, different configurations where tried. The algorithm did not show significant performance drop when varying G. The study for varying N can be seen in fig. 7. The most important result is that the attack remains completely successful with increasing complexity of the system –i.e. for large N–: it takes less time for the attacker to guess the internal state of A than to the machines to synchronize and the success of the attacks is strictly 1 for $N > 6$, thus A and B can not use PPMs as a cryptographic means. Another salient feature is the performance drop for low N. This might be due to correlations which are massively introduced in the system in this case, and with which the algorithm

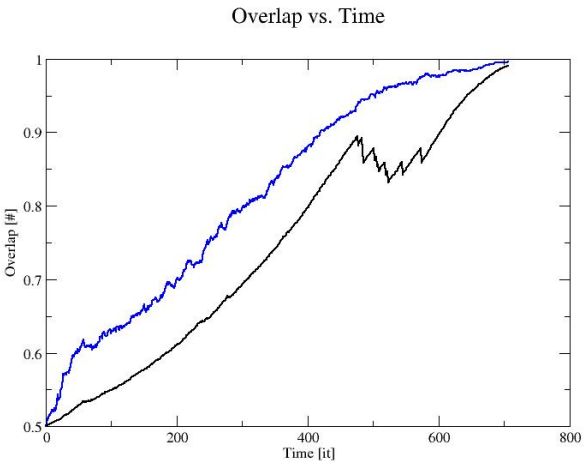

FIG. 6. **Performance of the Monte Carlo guessing for $K = 2$ after 3 *outer rounds*.** Black: averaged overlap. Blue: overlap of the most probable state vector. The time is measured in *inner rounds* and the results are averaged over 10 simulations, therefore, there are only data available as long as none of the simulations reached an end (they have different number of *inner rounds* per *outer round*). The settings of the PPM are $N = 8$, $K = 2$ and $G = 128$.

can not deal successfully. It remains, though, some information to be extracted from each outer round –as explained above and in fig. 5–, so the algorithm could be further improved. However, the interesting case for neural cryptography is that of increasing complexity.

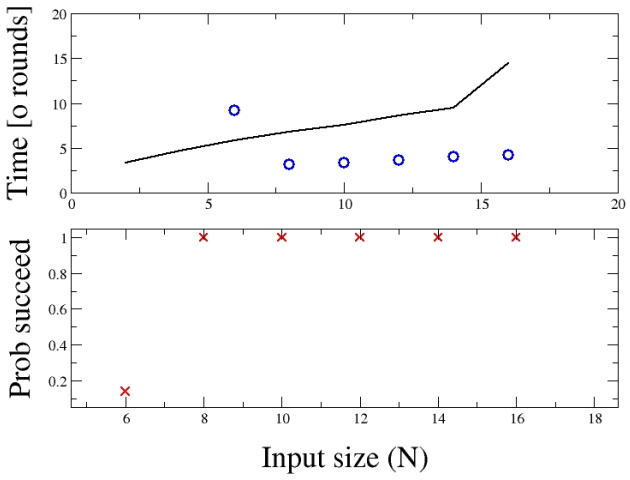

FIG. 7. **Performance of the Monte Carlo guessing for $K = 2$ with different input size**. **(a)** black line: synchronizing time of the PPMs A and B averaged over 100 realizations; blue circles: time that the algorithm must be applied to guess s_A completely. **(b)** probability that the attack is successful. Should be noted the drop of the performance for low N, but the algorithm remains completely effective with increasing complexity.

DISCUSSION

As pointed out in the introduction to this paper, guessing the state vector of a PPM based on a series of examples for synchronization (input vectors $\overrightarrow{x}$ and output of the machine τ_A) is a problem with some resemblance to the *mastermind* game. Here, less information is available than in the game and the alphabet is smaller (two bits in the PPM vs. a set of colours). In the field of *neural cryptography*, the only interest for the case $K = 1$ is that a method able of breaking more complicated systems ($K > 1$) should perform at least very well for this simple case. Therefore, PPM with $K = 1$ were used as working benches for promising new attacks on neural cryptographic systems.

The three methods described here showed fine or outstanding performance for $K = 1$. The method performing worst was a probabilistic Mean Field attack, which might be constrained by the hypothesis done for its analytical develop or the degeneracy of the keywords that the PPM can generate.

These are not important burdens for the Genetic and the Monte Carlo attacks described in this work. Both methods take some chances when guessing $\overrightarrow{s_A}$, which somehow avoid the constrain that affected the Mean Field approach. Both methods showed an outstanding performance, making it possible to guess a great percentage of the state vector of a PPM within a finite time in many cases. Some evidence is also presented that the full capacity of the methods has not been used. The good performance of these algorithms made them candidates to attack PPMs with $K = 2$.

Remaining in the problem with $K = 1$, the results obtained imply –in the case of the *mastermind* game– that the information available seems enough to solve the problem by mathematical means different from intuition (which a human player would use in the first place). The degeneracy of the solutions shows a real limitation of strictly analytic approaches. Since solutions which take chances (such as Monte Carlo and the Genetic Algorithm) avoid this barrier, we can still claim that there must be a mechanism –if not the human intuition then a similar one– which makes a bet on one subset of possible solutions, also risking that the right solution will be left out. We see also how a mechanism to correct this cases (e.g. mutation in the case of the Genetic Algorithm and the reset of some p_i in the case of Monte Carlo) must be implemented.

The case with real interest for *neural cryptography* is that with $K = 2$. Each method had a problem which impeded it from working even fine, but a clever combination of the Monte Carlo attack and an analytical calculus of the maximal-a-posteriori p_i^+ showed an outstanding performance in guessing the state vector of two synchronizing PPM with $K = 2$.

Also, the performance did not drop when increasing complexity was considered. The combined algorithm was able to break the protocol in 100% of the cases for $N > 6$ and for all the analyzed cases with varying G (not shown). It remains, though, a study of the method for odd N. This case is, anyway, also not so suit for cryptography since the synchronization is already quite slow.

It is also reported an important flow of the algorithm when breaking machines with $N \leq 6$. In these cases, again some neglected correlations could be hindering the performance and the algorithm does not work. It remains, though, some information that was not extracted and which the method could access, thus empowering the attack. Also, the case with low N is of lesser importance for cryptography.

A last remark about the probabilistic attack would be about its prospective application to TPM. Not only breaking the TPMs would be interesting: also, TPMs work based on learning algorithms. So finding an algorithm which *leanrs* faster than Hebbian learning could be of great interest also for artificial intelligence.

[1] Interacting Neural Networks.
R. Metzler, W. Kinzel, and I. Kanter,
Phisica A **33**, L141-L147, (200).

[2] Theory of Interacting Neural Networks.
W. Kinzel,
cond-mat/0204054.

[3] Secure exchange of information by synchronization of neural networks.
I. Kanter, W. Kinzel, E. Kanter,
Europhys. Lett. **57**, 141 (2002).

[4] Cryptography based on neural networks - analytical results.
M. Rosen–Zvi, I. Kanter, and W. Kinzel,
J. Phys. A; Math. Gen. **35**, L707, (2002).

[5] Analysis of Neural Cryptography.
A. Klimov, A. Mityaguine, and A. Shamir,
Advances in Cryptology—ASIACRYPT 2002, 288 (2003).

[6] Neural cryptography with queries.
A. Ruttor, W. Kinzel, and I. Kanter,
J. Stat. Mech. , P01009, (2005).

[7] The Theory of Neural Networks and Cryptography.
I. Kanter and W. Kinzel,
Quantum Computers and Computing **5** (1), 130-140, (2005).

[8] A. Ruttor.
Neural Synchronization and Cryptography.
Diss. Julius-Maximilians-Universität Würzburg, Würzburg 2006. Print.

[9] Permutation parity machines for neural cryptography.
O. M. Reyes and K.-H. Zimmermann,
Phys. Rev. E **81**, 066117 (2010).

[10] Cooperating Attackers in Neural Cryptograpgy.
L. N. Shacham, E. Klein, R. Mislovaty, I. Kanter, and W. Kinzel,
Phys. Rev. E **69**, 066137 (2004).

[11] Genetic attack on neural cryptography.
A. Ruttor, W. Kinzel, R. Naeh, and I. Kanter,
Phys. Rev. E **73**, 036121 (2006).

[12] Gaussian Processes for Classification: Mean Field Algorithms.
M. Opper and O. Winther,
Neural Computation **12** (11), 2655-2684, 2000.