Rainbow attack

Hans-Kristian Jørgensen
Renata Norbutaite

December 16, 2007

Introduction

In this paper we are going to present the technique known as rainbow tables. We will describe different kinds of trade-offs but in particular focus on time memory trade-offs. The historical development of the rainbow tables will be gone through where we start by Hellman's traitional time memory trade-off. We will describe the structure of rainbow tables and explain why and how they work. Finally we will conclude on what this technique can contribute with and where it fails.

Contents

1 Rainbow tables

In order to understand rainbow tables as a time/memory trade-off it is reasonable to begin by describing brute-force and dictionaries. Brute-force and dictionary attacks are the two extremes in an attack based on simple exhaustive search. When commencing a brute-force attack the adversary simply tries all possible combinations or keys and thus the chance of succeeding is 1, but given the time and computing power of the adversary this kind of attack is often thought of as infeasible as the average number of keys needed to succeed is half the total number of keys.

In the other extreme we have the dictionary attack. In the dictionary attack we precompute all possible keys and just look up the encrypted value to get the plaintext. This attack is almost instantanious but requires an enormous amount of memory as all possible plaintexts and ciphertexts needs to be stored. The needed storage to perform a dictionary attack is often to huge for this kind of attack to feasable.

The rainbow tables is an attempt to do a trade-off such that the memory needed is not as much as for a dictionary attack and the time needed is not as much as for a brute-force attack, thus the rainbow tables can be though of as a time/memory trade-off.

1.1 Vulnerable systems

Rainbow attacks can be used against many different systems, but one of the most vulnerable system is the Windows LanManager hash. This system is a special case as it is using capital letters only, truncated to two blocks of seven charecters and then uses DES on each of these seperate blocks. As has been the case with numerous applications of cryptosystems the LanManager weakness is not in the underlying principles but in the way it is put together. We will go through this particular system later in order to explain the principles used and how rainbow tables contribute to easy breach of the LanManager hash.

Of course there are other systems that are vulnerable to rainbow attacks for instance systems based on MD-class hashes and systems with predictable salts.

2 History

2.1 Hellman's method

In 1980 the first cryptanalytic time/memory trade-off was suggested by Hellman [4] [3] - a method to trade memory against attack time. For a cryptosystem having N keys, this method can recover a key in $N^{\frac{2}{3}}$ operations using $N^{\frac{2}{3}}$ words of memory. This method could be typically used to recover a key when the plaintext and the ciphertext are known. Now let's say we have a plaintext P and the corresponding ciphertext C, so the method tries to find a key $k \in N$ which was used to encrypt the plaintext with an encryption function E_k. That is: $C = E_k(P)$.

To use this method effectively we have to do a lot of work in advance. First, we have to try to generate all possible ciphertexts with all possible keys N. Then all the ciphertexts must be organized in chains so that only the first and the last element of the chain are stored in the memory. This is where we get the trade-off concept. We are saving memory here at the cost of cryptanalysis time. The chains are created using a reduction function R, which creates a key from a ciphertext. By applying encryption function E_k and the reduction function R, we can thus create chains of alternating keys and ciphertexts $k_i \xrightarrow{E_k(P)} C_i \xrightarrow{R(C_i)} k_{i+1}$.

This can be rewritten as $R(E_k(P)) = f(k)$ and so using this function we are generating a key from a key and at the end of the day we get chains of keys: $k_i \xrightarrow{f(k_i)} k_{i+1} \xrightarrow{f(k_{i+1})} k_{i+2} \xrightarrow{f(k_{i+2})} ...$

Let's say we create m chains of length t. As it was mentioned before only the first and the last element are stored in the table. Now, if we are given a ciphertext C we can try to find out whether the key used in the encryption is one of the keys used to generate the table. To check this we generate a new chain of keys this time starting with $R(C)$ where C is a given ciphertext. The chain we generate is up to length t. If $C = E_k(P)$ where k was one of the keys used to generate our table, then we eventually generate the key that matches the last key of a corresponding chain (the last key which we are saving in the table we generated earlier). Now using the first key (corresponding to the last we just found) the whole chain can be regenerated. The key we are actually looking for will be the one just before $R(C)$ as this was the key used for the encryption that lead to the particular ciphertext.

The fundamentals of this method seem to be quite easy, but let's look at the wider picture. Hellman's scheme consists of several tables where each

of them covers only a small portion of the possible values of $f(k)$. A lower bound on the expected coverage of images by a single table in his scheme was calculated and Hellman's analysis of the coverage of images by the full scheme was highly heuristic, but showed a success rate of this trade-off for a random f is about 55 %. Later Shamir and Spencer proved in a single table way that for a majority of the functions f, even the best Hellman table has about the same coverage of images as a random Hellman table, although this was not proven for multi-table Hellman scheme.

2.2 Problems with Hellman's scheme

It may happen that even if the chains start with different keys they can collide and merge [4]. This is due to the reduction function R which is arbitrary when going from the space of ciphertexts into the space of keys. Therefore, the larger table we have the higher probability we have that this would happen. Each merge reduces the number of distinct keys which suppose to be covered in the table. We are given the success rate to find a key in a single table, when there is m rows and t keys:

$$P \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(1 - \frac{i \cdot t}{N}\right)^{j+1}$$

So to have better chance to find a key we need, it is reasonable to use many tables instead of one, where each of them use a different reduction function. Because of this, chains of different tables can collide but they will not merge. We are given the success rate to find a key when we have l tables:

$$P \geq 1 - \left(1 - \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(1 - \frac{i \cdot t}{N}\right)^{j+1}\right)^{l}$$

2.3 False alarms

When describing the original method, we mentioned that all that is necessary to be done is to generate the key that matches the last key which we are saving in the table we generated earlier. But that is not always the case. Even if we match the last key that gives no absolute guarantees that we will find the key in that chain or in that table. It might be that the key is just a part of a chain with the same last key, but regenerating the chain from the first

key will not give us the key we are looking for. And this situation is called a false alarm.

A different kind of a false alarm can occur when there is a key in a chain that is a part of a table but it merges with other chains of that table. Then there are some different first keys corresponding to the same last key.

Although Hellman estimated that the expected computation due to false alarms increases the expected computation by at most 50 %, this is not exactly the case. His reasoning relied on the fact that for any i, $f^i(k_1)$ is computed by iterating fi times. However $f^i(k_1)$ should be computed from k_1, because $f^i(k_1) = f(k_i)$. In this case the computation time required to reach a chains end is significantly reduced on average while the computation time required to rule out false alarms stays the same [3]. Therefore, false alarms can increase the expected computation time by more than 50 %.

2.4 Optimization by Rivest

L. Rivest introduced an optimization based on distinguished points to detect matching end point of a table [3]. Distinguished points are keys (or ciphertexts) that satisfy a given criterion, e.g. the last n bits are all zero. In this variant chains are not generated with a given length but they stop at the first occurrence of a distinguished point. So instead of looking up in the table each time a key is generated on the chain from C, keys are generated until a distinguished point is found and only then a look-up is carried out in the table. If the average length of the chains is t, then this optimization reduces the amount of look-ups by a factor t.

2.5 Advantages of distinguished points and rainbow tables

- As it was mentioned before in L. Rivest optimization, the number of table look-ups is reduced by a factor of t compared to Hellman's original method [4].
- Merges of rainbow chains results in identical end points and are thus detectable as with distinguished points. Rainbow chains can thus be used to generate merge-free tables (but not collision free).

- Rainbow chains have no loops, since each reduction function appears only once. This is better than loop detection and rejection, because we don't spend time on following and then rejecting loops and the coverage of our chains is not reduced because of loops than can not be covered.
- Rainbow chains have a constant length whereas chains ending in distinguished points have a variable length. This reduces the number of false alarms and the extra work due to false alarms. This effect can be much more important than the factor of two gained by the structure of the table.

3 Structure of the rainbow tables

One of the most important improvements that Oechslin did with his rainbow tables was to drastically diminish merging of chains [4]. In Hellman's tables two chains would merge if they collided as the reduction function remained the same throughout the table. In Oechslins rainbow tables the reduction function is changed for each link which means that in order to merge the two chains must have a collision at the exact same position in the chain. The possibility of a merge of a collicion when the length of the chains are t is of course $\frac{1}{t}$.

The construction of a rainbow table is similar to that of a traditional Hellman table but with a few differences. What we need is a list of all possible passwords and their hashes so we start by choosing a random password within the set of possible passwords. It could for instance be limited to passwords with the pattern $[A-Za-z0-9]\{6,8\}$ which would include most typical used passwords of short length (6-8 characters). Then we perform the one-way function (hash function) and get an output which we in turn use as input in what is called the reduction function. The reduction function then generates a new password based on the input which is send into the one-way function and so on and so forth until the we have reached the t'th link in the chain. Finally we store the beginning of the chain and the end of it both being a possible password.

This is similar to the generation of the Hellman tables, but the before mentioned reduction function changes for each link in the chain. This means that instead of seing the reduction function as $password = R(hash)$ it is changed into $password = R(hash, number\ in\ chain)$. This is really important as that is what grants the desired ability of the table; diminish merges.

In the classical Hellman tables two chains will merge when the same value is output from the same function at some point in the chain. This is because the output of the following functions will be the same as input is the same. To avoid this merging, an extra parameter is introduced and as the function is based on both inputs they need both by the same in order to produce the same output. This is why we have the possibility of a merge of $\frac{1}{t}$ when we have a collision of hashes as stated earlier.

3.1 Why fixed length?

As mentioned in 2.5 rainbow tables benefit from being of constant length in contrast to the optimisation proposed by Rivest in 1982 where the distinguished points results in chains with variable length. This is because of two important properties that Oechslin names *fatal attraction* and *larger overhead* [4].

Fatal attraction covers the fact that the longer the chain the higher the probability of a merge is and equally important - the more merges the more chains needs to be checked. Consider a short chain with few links and a long chain with many links. If we randomly choose values there is of course higher probability of choosing a value in the long chain than in the short one thus it is obvious that longer chains will have more merges than shorter chains. In case of a false alarm[1] we need to check all the beginnings that might lead to the matching ending which means that if we have more merges we need to check more chains and thus extends the cost of a false alarm. Combined with the fact that the longer chains has more merges and that we are interested in having long chains in order to achieve the reduction of memory requirement, we have that chains of varying length are costly because they come with the property of *fatal attraction*.

The larger overhead means that in chains with variable length we do not know when we will reach the end of it and thus we need to run through the chain until we reach the end. In rainbow tables you have fixed length and different reduction functions that you use thus you know exactly when you are supposed to find the value you search for. This means that you do not have to perform as many calculations as in the tables with chains of variable length. Another characteristic of the rainbow tables is that the longer chain you need to build in order to find a matching ending, the shorter the remaining part of the chain that you need to check is, and as there is

[1] See p. 4

a greater chance of false alarms in the longer chain there is a higher chance that we need not to check a long fraction of the chain.

3.2 The success rate

When calculating the success rate of rainbow tables we can look at one column of the table at the time and treat it as a classical occupancy problem. The occupancy problem is the problem where we are having n bins and m balls and we want to distribute the balls in the bins. It easy to find out whether all balls have been distributed and whether the number of bins equals the number of balls, but in our case we need to find out whether all balls (possible keys) are present in the bins (tables).

We start with $m_1 = m$ distinct keys in the first column. In the second column the m_1 keys are randomly distributed over the keyspace of size N, generating m_2 distinct keys [4]:

$$m_2 = N(1 - (1 - \frac{1}{N})^{m_1}) \approx N(1 - e^{-\frac{m_1}{N}})$$

As each column i has m_i distinct keys, the success rate of the entire table is:

$$P = 1 - \prod_{i=1}^{t} (1 - \frac{m_i}{N})$$

where t is the length of a chain, $m_1 = m$ and $m_{n+1} = N(1 - e^{-\frac{m_n}{N}})$

This is the probability that none of the N keys are not represented in the table or in other words the probability that we have covered all N keys.

The same approach can be used to calculate the number of non-merging chains that can be generated. Since merging chains are recognised by their identical endpoint, the number of distinct keys in the last column m_t is the number of non-merging chains. The maximum number of chains can be reached when choosing every single key in the key space N as a starting point [4].

The success probability of a table with the maximum number of non-merging chains is:

$$m_1 = N \quad , \quad m_{n+1} = N(1 - e^{-\frac{m_n}{N}})$$

It should be noted however that the effort to build such a table is Nt.

3.3 How to use the table

After having built the table we can perform an attack that is close to the traditional dictionary attack, but because of our trade-off we can not just look up any given value in the table and get the key. Instead we have to perform a set of operations in order to reconstruct the parts that we need. As the hash function is a one way function and the reduction function has been designed likewise we can not just go backwards in the chain, we have to generate from earlier in the chain.

When we need to lookup a value in the table we first apply R_{n-1}, the last reduction function used in the construction of the rainbow table, to the ciphertext. We now have a hash value that can be compared to the values in the last column of hashes in the table and thus if this hash value has a collision in the last column we have found the chain we need to regenerate.

What we actually do is to start at different positions within the chain and run the functions that were run in the generation of the table on the value that we want to look up. We start by generating the values that can be compared to the ending points, but if a collision is not found we go one step further into the chain and thus instead of performing R_{n-1} we perform R_{n-2}, f_{n-1} where f_{n-1} is the pair of functions consisting of H_{n-1} and R_{n-1}.

If we have still not found a match in the ending column we repeat this method by applying $R_{n-3}, f_{n-2}, f_{n-1}$ and so on until we either have a match or we have reached the starting values without having a match. In the latter case the value is not present in the table but in the prior we still have a few operations to perform in order to find the key.

When we have found a match we need to regenerate the chain where we found the match. This is done by using the starting value of the chain and then simply run the set of functions again. When we reach the ciphertext within the chain the chain generation we know that the previous value is the key that we were looking for and thus this value is returned.

4 Trade-offs

4.1 Time/memory trade-offs

As we know, Hellman proposed a cryptanalytic time memory trade-off approach which reduces cryptanalysis time by using a pre-computed table

stored in memory. By defining as M is required amount of memory, T is the operations in time and N is possible solutions to search over. This search space is expressed in terms of M and T as: $M^2T = N^2$ [2][1].

In this scheme D=1 (where D is considered to be an amount of known data). The table preparation time is disregarded and only the online time and memory requirements are considered. The assumption is that tables would be prepared once for all in an offline phase. Once the tables are prepared, they will not change and can be used to find different pre-images. In this scenario, the table preparation time can be huge and even larger than exhaustive search. Thus, the security of a cryptographic algorithm with respect to this kind of time/memory trade-off has a hidden cost of offline (and one time) exhaustive search.

4.2 Time/memory/data trade-offs

Hellman's algorithm has been generalized for the case of multiple data [1], which resulted in the formula $N^2 = TM^2D^2$, $1 \leq D < T$, $P = \frac{N}{D}$. The algorithm consists of two stages: a one-time offline stage followed by an online stage. In the online stage, we will be given D points $y_1, y_2, ...y_D$ in the range of f and our goal is to find the pre-image of any one of these points.

In the offline stage, a set of tables are prepared covering $\frac{N}{D}$ of the domain points. So, if the multiple data is available, the actual table preparation time will be less than exhaustive search. Since this is an offline activity, it might be reasonable to expect the table preparation time to be more than the online time but less than exhaustive search.

The pre-computation time will be in general more than the memory requirement. In the table preparation stage, the entire table will have to be computed and only a fraction of it stored. This shows that the offline time will be at least as large as the memory requirement. Hellman considered the condition where the online time is equal to the memory requirement. In the presence of multiple data, it is perhaps more practical to require the data and memory requirement to be less than the online and offline time requirements. A possible adaptation for the Rainbow scheme to time/memory/data resulting curve of $N^2 = TM^2D$ is far inferior to the curve presented for Hellman's method. There are some ways of implementing Rainbow based on these trade-offs.

The first method is to reduce the number of colors to S (where S is the number of values that the hidden state can assume and by *colors* we mean different reduction functions) by repeating the series of colors t times. The resulting matrix is called *thin-Rainbow* matrix. The resulting trade-off is $N^2 = TM^2D^2$, which is similar to the trade-off of Hellman's method. It is slightly inferior to that, this method requires twice as many bits to represent its start points. In the online phase, each color is sequentially tried by continuing the chain for at most tS links.

For the second method, consider a scheme in which we group colors together in groups of t, and the resulting matrix is a *thick-Rainbow* matrix. During the online phase the algorithm needs to guess not only the "flavor" i of f_i, but also the phase of f_i among other f_i's. The hidden state is larger than S and includes the phase. And the phase affects the development of the chain. Thus we get an inferior trade-off $N^2 = TM^2D$.

When using distinguished points, we can not only determine the end of the chain, but also to determine the points in which we switch from one flavor of f to another. In this case, the number of hidden states is equal to the number of flavors, and does not have to include any additional information. The resulting matrix is called a *fuzzy-Rainbow* matrix, as each hidden state appears in slightly different locations in different rows of the matrix. In the online phase, the colors are tried in the same order as in the Rainbow scheme. Analysis shows that the trade-off curve is $2TM^2D^2 = N^2 + ND^2M$ with $T \geq D^2$.

4.3 Time/memory/key trade-offs

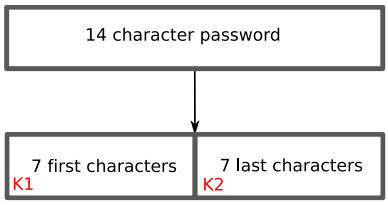
Let's use $\frac{t}{D_k}$ tables instead of t [1], which means that memory requirements go down to $M = \frac{mt}{D_k}$ (where m is the number of Hellman tables). The trade-off formula for this situation is $N^2 = T(MD_k)^2$. It is important to note that unlike in Hellman's original trade-off, the pre-processing time is much lower than the exhaustive search and thus technically this is a break of a cipher.

5 The original experiment

The original experiment with the purpose to show the efficiency of the rainbow tables was performed on the LanManager. As mentioned in 1.1 this

particular system has a group of characteristics that makes it a perfect system to attack with rainbow tables. The LanManager is still available in Vista for backward compatibility, but it is switched off by default. This however is not the case with previous versions where you need to disable it yourself.

The LanManager is generated by taking a 14 character password and chop it into two chunks of 7 characters each:



In cases where the password originally was longer than 14 characters it is truncated to 14 and if it is shorter it is padded with 0x00 bytes [2]. K1 and K2 are then converted to uppercase and the resulting 56 bit string is then used as a key in DES. The plaintext encrypted is a *magic 8 byte number* or more specific **KGS!@#$%**. The plaintext is encrypted with K1 and the output is concatenated with a similar encryption of the same plaintext but with K2 as the key. This construction means that in practice we have two blocks of 8 bytes each, as the output of DES is 8 bytes, instead of the a 16 byte block resulting in a keyspace of 2^{37} instead of 2^{83}.

Multiple experiments have been performed on the LanManager hash and all have shown the exact same result; the rainbow attack is extremely efficient at short non-salted passwords such as the implementation in LanManager. What is rather interesting though is that the most important property of rainbow tables is not the decrease in number of table look-ups compared to the original trade-off, but the fact that the number of merges and false alarms are reduced. This is due to the characteristics mentioned in 3.1.

6 The results of the LM experiment

Let us look at the results of the Lan Manager attack performed by Oechslin [4]. Say we have the parameters for classic tables: length of a chain is $t_c = 4666$ and there is $m_c = 8192$ chains per table. Parameters for a rainbow table: length of a chain is $t_r = 4666$ and there is $m_r = t_c m_c = 38223872$ chains per

[2] `http://pubwww.tudelft.nl/swat/help/ENCRYPTION.html`

table. There has been generated 4666 classic tables and one rainbow table. Their success were rated by cracking 500 random passwords on a workstation with parameters: P4, 1.5 GHz, 500MB RAM.

The experiment showed that the rainbow table achieved the same success rate with the same amount of data as classical tables. They were compared on mean cryptanalysis time, the mean number of hash operations per cryptanalysis and the mean number of false alarms per cryptanalysis. The result was that the rainbow method was 7 times faster than the original method.

Why not try to use more than one rainbow table? They chose to generate 5 tables of 35000000 lines in order to respect the memory constraint of 1.4 GB. As for the classic tables, there were 23330 of them with 4666 columns and 7501 lines. The success was rated by cracking 500 random passwords and in both cases the success rate was 99.9 %. Rainbow tables need 12 times less calculations than classical tables with distinguished points. Unfortunately the gain in time is only a factor of 1.5.

7 The results of the Unix experiment

Let us take a look at another experiment, the Unix attack [1]. Her we use time/memory/data trade-off to analyze Unix password scheme. Say there is $D = 1000$ password hashes and trade-off space consists of 56-bits of the unknown password and 12-bits of known salt. Since the salt size is much shorter than the key-size, its effect on making trade-off harder is not very significant.

Say the attacker knows which symbols are allowed to form a password and he knows that it can be encoded in 48-bits. Together with the salt we have: $N = 2^{60}$.

The attack parameters: preprocessing time is $P = N/D = 2^{50}$, but that is done only once. Memory of $M = 2^{34}$ 8-byte entries (12+48 bits) which takes one 128 GB hard-disk. The attack time $T = 2^{32}$. That would be about an hour for a fast PC. The attack would recover one password from about every 1000 new password hashes supplied.

Such a trade-off could analyze all passwords typable on a keyboard. The space is $N = 2^{63}$. Assuming again $D = 2^{10}$, we get pre-computation time $P = 2^{53}$, $M = 2^{35}$ 8-byte entries or one 256 GB hard disk, $T = 2^{36}$ hash evaluations.

8 The generalised version

As mentioned the rainbow tables has traditionally been constructed for the LanManager hash but Zhu Shuanglei has released a generalised rainbow table generator known as *rainbowcrack*. All that is done is to change the keyspace, the encryption function and the reduction function, but this opens up for a wider usage for instance tables to break MD5 and SHA-1 can easily be built with this program. The breaking of MD5, SHA-1 or NTLM requires way more pre-computational time and memory, but what do you do then if you want to create a such table but do not have the capacity yourself?

There are many communities around the internet where people have joined the task of creating huge rainbow tables and share the efforts with each other. For instance there is `http://wiki.hak5.org/wiki//Community_Rainbow_Tables`, `http://rainbowcrack.com` and `http://www.freerainbowtables.com/`. If you want the rainbow tables to be positioned at your own computer, the Shmoo group has even released their generated rainbow tables at `http://rainbowtables.shmoo.com/`.

9 Preventing the attacks

Although there was analyzed a trade-off on a Unix password scheme [1] it was assumed that salt is known. A rainbow table is ineffective against one-way hashes that include unknown salts [6]. If the rainbow tables do not have passwords matching the length and complexity of the salted password, then the password will not be found. But if it actually is found, one will have to remove the salt from the password before it could be used. To prevent against a site specific rainbow table it is important that salts are stored per user, otherwise an attacker can just create a rainbow table including the salt of the site and possible passwords, which for a large site would produce a number of hits.

So it is clear, that these methods can only be applied if all possible hashes can be calculated in advance [5]. This is not possible in most operating systems, since every password hash is calculated with a different recipe, adding a random amount of salt. This salt is stored together with the hash such that a password can later be verified to match the hash. But despite of that since we do not know in advance which value of salt will be used with the hash we want to crack, we cannot create rainbow tables in advance.

In general, rainbow tables tend to have little or no success when trying to work outside the range of symbols or password length computed into the table. Choosing a password that is longer or contains symbols not accounted for inside a rainbow table can be very effective. Because of the sizable investment in computing processing, rainbow tables beyond 8 places in length are not yet common.

10 Conclusion

In this paper we have examined rainbow tables from their origin in Hellman's original time/memory trade-off over their construction, usage, strength and weaknesses. We have looked at different experiments with rainbow tables and found that the usage of rainbow tables might not have been entirely removed by the introduction of salting.

The main contribution of the rainbow tables are the improved structure of apparently random data and in particular a solution to the merging problem. As time/memory trade-offs benefit 4 times from Moore's law this is still an interesting technique when combining with distributed computing. Even though you could increase the keyspace and thereby increasing both the precomputation and memory requirements significantly, techniques like rainbow tables have made brute-force attacks interesting in large scale.

It is also a bit scary to see that many actual implementations of theoretically secure cryptosystems has proven to be insecure. For instance the LanManager and Oracle's implementations where even though salt is used in Oracle it is predictable[3] and thus can be included in the generation of the rainbowtable.

[3]The password is prepended with the username and the admin account is named system

References

[1] Sourav Mukhopadhyay Alex Biryukov and Palash Sarkar. Improved time-memory trade-offs with multiple data. *SAC 2005, LNCS 3897, pp. 110-127*, 2005. 10, 11, 13, 14

[2] Eli Biham Elad Barkan and Adi Shamir. Rigorous bounds on cryptanalytic time-memory trade-offs. *CRYPTO 2006, LNCS 4117, pp. 1-21*, 2006. 10

[3] Pascal Junod Gildas Avoine and Philippe Oechslin. Time-memory trade-offs: False alarm detection using checkpoints. *INDOCRYPT 2005, LNCS 3797, pp. 183-196*, 2005. 3, 5

[4] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. *Laboratoire de Securite et de Cryptographie (LASEC)*, 2003. 3, 4, 5, 6, 7, 8, 12

[5] Philippe Oechslin. Password cracking: Rainbow tables explained. *Constituent Contributions*, 2005. 14

[6] wikipedia. Rainbow tables. `http://en.wikipedia.org/wiki/Rainbow_table`, 2007. 14