Making a Faster Cryptanalytic Time-Memory Trade-Off

Philippe Oechslin

Laboratoire de Securité et de Cryptographie (LASEC)
Ecole Polytechnique Fédérale de Lausanne
Faculté I&C, 1015 Lausanne, Switzerland
philippe.oechslin@epfl.ch

Abstract. In 1980 Martin Hellman described a cryptanalytic time-memory trade-off which reduces the time of cryptanalysis by using precalculated data stored in memory. This technique was improved by Rivest before 1982 with the introduction of distinguished points which drastically reduces the number of memory lookups during cryptanalysis. This improved technique has been studied extensively but no new optimisations have been published ever since. We propose a new way of precalculating the data which reduces by two the number of calculations needed during cryptanalysis. Moreover, since the method does not make use of distinguished points, it reduces the overhead due to the variable chain length, which again significantly reduces the number of calculations. As an example we have implemented an attack on MS-Windows password hashes. Using 1.4GB of data (two CD-ROMs) we can crack 99.9% of all alphanumerical passwords hashes (2^{37}) in 13.6 seconds whereas it takes 101 seconds with the current approach using distinguished points. We show that the gain could be even much higher depending on the parameters used.

Key words: time-memory trade-off, cryptanalysis, precomputation, fixed plaintext

1 Introduction

Cryptanalytic attacks based on exhaustive search need a lot of computing power or a lot of time to complete. When the same attack has to be carried out multiple times, it may be possible to execute the exhaustive search in advance and store all results in memory. Once this precomputation is done, the attack can be carried out almost instantly. Alas, this method is not practicable because of the large amount of memory needed. In [4] Hellman introduced a method to trade memory against attack time. For a cryptosystem having N keys, this method can recover a key in $N^{2/3}$ operations using $N^{2/3}$ words of memory. The typical application of this method is the recovery of a key when the plaintext and the ciphertext are known. One domain where this applies is in poorly designed data encryption system where an attacker can guess the first few bytes of data (e.g.

"`#include <stdio.h>`"). Another domain are password hashes. Many popular operating systems generate password hashes by encrypting a fixed plaintext with the user's password as key and store the result as the password hash. Again, if the password hashing scheme is poorly designed, the plaintext and the encryption method will be the same for all passwords. In that case, the password hashes can be calculated in advance and can be subjected to a time-memory trade-off.

The time-memory trade-off (with or without our improvement) is a probabilistic method. Success is not guaranteed and the success rate depends on the time and memory allocated for cryptanalysis.

1.1 The original method

Given a fixed plaintext P_0 and the corresponding ciphertext C_0, the method tries to find the key $k \in N$ which was used to encipher the plaintext using the cipher S. We thus have:

$$C_0 = S_k(P_0)$$

We try to generate all possible ciphertexts in advance by enciphering the plaintext with all N possible keys. The ciphertexts are organised in chains whereby only the first and the last element of a chain is stored in memory. Storing only the first and last element of a chain is the operation that yields the trade-off (saving memory at the cost of cryptanalysis time). The chains are created using a *reduction function* R which creates a key from a cipher text. The cipher text is longer that the key, hence the reduction. By successively applying the cipher S and the reduction function R we can thus create chains of alternating keys and ciphertexts.

$$k_i \xrightarrow{S_{k_i}(P_0)} C_i \xrightarrow{R(C_i)} k_{i+1}$$

The succession of $R(S_k(P_0))$ is written $f(k)$ and generates a key from a key which leads to chains of keys:

$$k_i \xrightarrow{f} k_{i+1} \xrightarrow{f} k_{i+2} \rightarrow ...$$

m chains of length t are created and their first and last elements are stored in a table. Given a ciphertext C we can try to find out if the key used to generate C is among the ones used to generate the table. To do so, we generate a chain of keys starting with $R(C)$ and up to the length t. If C was indeed obtained with a key used while creating the table then we will eventually generate the key that matches the last key of the corresponding chain. That last key has been stored in memory together with the first key of the chain. Using the first key of the chain the whole chain can be regenerated and in particular the key that comes just before $R(C)$. This is the key that was used to generate C, which is the key we are looking for.

Unfortunately there is a chance that chains starting at different keys collide and merge. This is due to the fact that the function R is an arbitrary reduction

of the space of ciphertexts into the space of keys. The larger a table is, the higher is the probability that a new chain merges with a previous one. Each merge reduces the number of distinct keys which are actually covered by a table. The chance of finding a key by using a table of m rows of t keys is given in the original paper [4] and is the following:

$$P_{table} \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(1 - \frac{it}{N}\right)^{j+1} \tag{1}$$

The efficiency of a single table rapidly decreases with its size. To obtain a high probability of success it is better to generate multiple tables using a different reduction function for each table. The probability of success using ℓ tables is then given by:

$$P_{success} \geq 1 - \left(1 - \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(1 - \frac{it}{N}\right)^{j+1}\right)^{\ell} \tag{2}$$

Chains of different tables can collide but will not merge since different reduction functions are applied in different tables.

False alarms When searching for a key in a table, finding a matching endpoint does not imply that the key is in the table. Indeed, the key may be part of a chain which has the same endpoint but is not in the table. In that case generating the chain from the saved starting point does not yield the key, which is referred to as a false alarm. False alarms also occur when a key is in a chain that is part of the table but which merges with other chains of the table. In that case several starting points correspond to the same endpoint and several chains may have to be generated until the key is finally found.

1.2 Existing work

In [2] Rivest suggests to use distinguished points as endpoints for the chains. Distinguished points are points for which a simple criteria holds true (e.g. the first ten bits of a key are zero). All endpoints stored in memory are distinguished points. When given a first ciphertext, we can generate a chain of keys until we find a distinguished point and only then look it up in the memory. This greatly reduces the number of memory lookups. All following publications use this optimisation.

[6] describes how to optimise the table parameters t, m and ℓ to minimise the total cost of the method based on the costs of memory and of processing engines. [5] shows that the parameters of the tables can be adjusted such as to increase the probability of success, without increasing the need for memory or the cryptanalysis time. This is actually a trade-off between precomputation time and success rate. However, the success rate cannot be arbitrarily increased.

Borst notes in [1] that distinguished points also have the following two advantages:

- They allow for loop detection. If a distinguished point is not found after enumerating a given number of keys (say, multiple times their average occurrence), then the chain can be suspected to contain a loop and be abandoned. The result is that all chains in the table are free of loops.
- Merges can easily be detected since two merging chains will have the same endpoint (the next distinguished point after the merge). As the endpoints have to be sorted anyway the merges are discovered without additional cost. [1] suggest that it is thus easy to generate collision free tables without significant overhead. Merging chains are simply thrown away and additional chains are generated to replace them. Generating merge free tables is yet another trade-off, namely a reduction of memory at the cost of extra precomputation.

Finally [7] notes that all calculations used in previous papers are based on Hellman's original method and that the results may be different when using distinguished points due to the variation of chain length. They present a detailed analysis which is backed up by simulation in a purpose-built FPGA.

A variant of Hellman's trade-off is presented by Fiat and Noar in [3]. Although this trade-off is less efficient, it can be rigorously analysed and can provably invert any type of function.

2 Results of the original method

2.1 Bounds and parameters

There are three parameters that can be adjusted in the time-memory trade-off: the length of the chains t , the number of chains per table m and the number of tables produced ℓ.

These parameters can be adjusted to satisfy the bounds on memory M, cryptanalysis time T and success rate $P_{success}$. The bound on success rate is given by equation 2. The bound on memory M is given by the number of chains per table m, the number of tables ℓ and the amount of memory m_0 needed to store a starting point and an endpoint (8 bytes in our experiments). The bound in time T is given by the average length of the chains t, the number of tables ℓ and the rate $\frac{1}{t_0}$ at which the plaintext can be enciphered (700'000/s in our case). This bound corresponds to the worst case where all tables have to be searched but it does not take into account the time spent on false alarms.

$$M = m \times \ell \times m_0 \qquad\qquad T = t \times \ell \times t_0$$

Figure 1 illustrates the bounds for the problem of cracking alphanumerical windows passwords (complexity of 2^{37}). The surface on the top-left graph is the bound on memory. Solutions satisfying the bound on memory lie below this surface. The surface on the bottom-left graph is the bound on time and solutions also have to be below that surface to satisfy the bound. The graph on the right side shows the bound on success probability of 99.9% and the combination of the two previous bounds. To satisfy all three bounds, the parameters of the

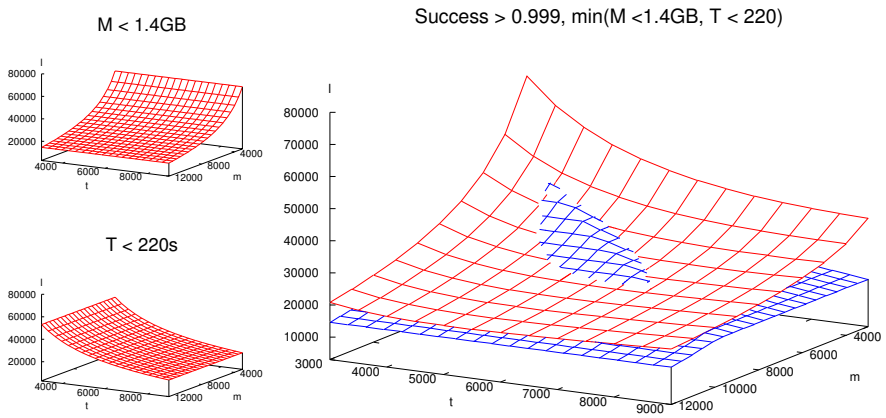

Fig. 1. Solution space for a success probability of 99.9%, a memory size of 1.4GB and a maximum of 220 seconds in our sample problem.

solution must lie below the protruding surface in the centre of the graph (time and memory constraints) and above the other surface (success rate constraint). This figure nicely illustrates the content of [5], namely that the success rate can be improved without using more memory or more time: all the points on the ridge in the centre of the graph satisfy both the bound on cryptanalysis time and memory but some of them are further away from the bound of success rate than others. Thus the success rate can be optimised while keeping the same amount of data and cryptanalysis time, which is the result of [5]. We can even go one step further than the authors and state that the optimal point must lie on the ridge where the bounds on time and memory meet, which runs along $\frac{t}{m} = \frac{T}{M}$. This reduces the search for the optimal solution by one dimension.

3 A new table structure with better results

The main limitation of the original scheme is the fact that when two chains collide in a single table they merge. We propose a new type of chains which can collide within the same table without merging.

We call our chains rainbow chains. They use a successive reduction function for each point in the chain. They start with reduction function 1 and end with reduction function $t-1$. Thus if two chains collide, they merge only if the collision appears at the same position in both chains. If the collision does not appear at the same position, both chains will continue with a different reduction function and will thus not merge. For chains of length t, if a collision occurs, the chance of it being a merge is thus only $\frac{1}{t}$. The probability of success within a single

table of size $m \times t$ is given by:

$$P_{table} = 1 - \prod_{i=1}^{t}(1 - \frac{m_i}{N}) \tag{3}$$

$$\text{where} \quad m_1 = m \quad \text{and} \quad m_{n+1} = N\left(1 - e^{-\frac{m_n}{N}}\right)$$

The derivation of the success probability is given in the appendix. It is interesting to note that the success probability of rainbow tables can be directly compared to that of classical tables. Indeed the success probability of t classical tables of size $m \times t$ is approximately equal to that of a single rainbow table of size $mt \times t$. In both cases the tables cover mt^2 keys with t different reduction functions. For each point a collision within a set of mt keys (a single classical table or a column in the rainbow table) results in a merge, whereas collisions with the remaining keys are not merges. The relation between t tables of size $m \times t$ and a rainbow table is shown in Figure 2. The probability of success are compared in Figure 3. Note that the axes have been relabeled to create the same scale as with the classical case in Figure 1. Rainbow tables seem to have a slightly better probability of success but this may just be due to the fact that the success rate calculated in the former case is the exact expectation of the probability where as in the latter case it is a lower bound.

To lookup a key in a rainbow table we proceed in the following manner: First we apply R_{n-1} to the ciphertext and look up the result in the endpoints of the table. If we find the endpoint we know how to rebuild the chain using the corresponding starting point. If we don't find the endpoint, we try if we find it by applying R_{n-2}, f_{n-1} to see if the key was in the second last column of the table. Then we try to apply $R_{n-3}, f_{n-2}, f_{n-1}$, and so forth. The total number of calculations we have to make is thus $\frac{t(t-1)}{2}$. This is half as much as with the classical method. Indeed, we need t^2 calculations to search the corresponding t tables of size $m \times t$.

Rainbow chains share some advantages of chains ending in distinguished points without suffering of their limitations:

- The number of table look-ups is reduced by a factor of t compared to Hellman's original method.
- Merges of rainbow chains result in identical endpoints and are thus detectable, as with distinguished points. Rainbow chains can thus be used to generate merge-free tables. Note that in this case, the tables are not collision free.
- Rainbow chains have no loops, since each reduction function appears only once. This is better than loop detection and rejection as described before, 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. As we shall see in Section 4.1 this

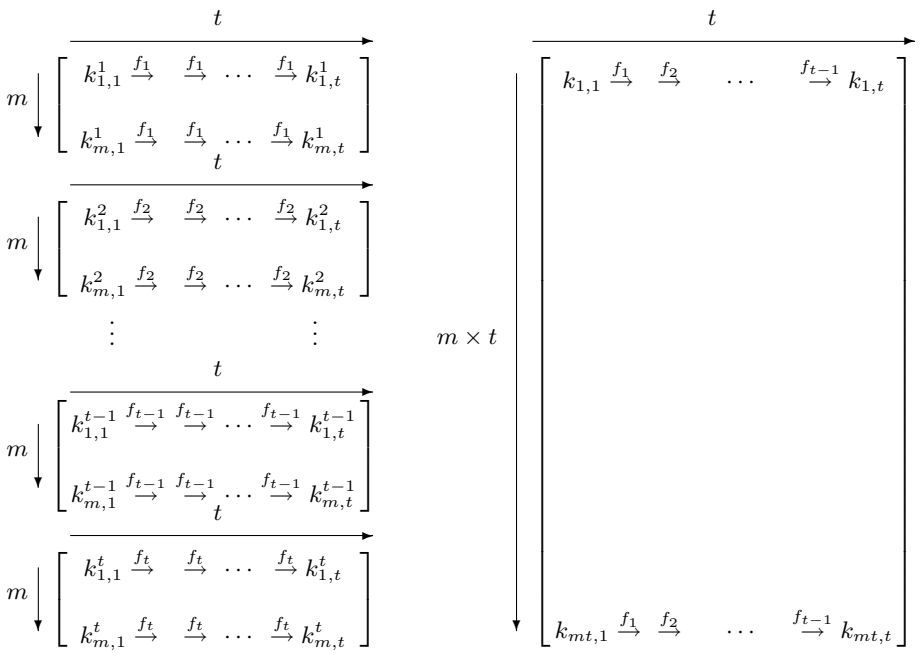

Fig. 2. t classic tables of size $m \times t$ on the left and one rainbow table of size $mt \times t$ on the right. In both cases merges can occur within a group of mt keys and a collision can occur with the remaining $m(t-1)$ keys. It takes half as many operations to look up a key in a rainbow table than in t classic tables.

reduces the number of false alarms and the extra work due to false alarms. This effect can be much more important that the factor of two gained by the structure of the table.

4 Experimental results

We have chosen cracking of MS Windows passwords as an example because it has a real-world significance and can be carried out on any standard workstation. The password hash we try to crack is the LanManager hash which is still supported by all versions of MS Windows for backward compatibility. The hash is generated by cutting a 14 characters password into two chunks of seven characters. In each chunk, lower case characters are turned to upper case and then the chunk is used as a key to encrypt a fixed plain-text with DES. This yields two 8 byte hashes which are concatenated to form the 16 byte LanManager hash. Each halves of the LanManager hash can thus be attacked separately and passwords of up to 14 alphanumerical generate only 2^{37} different 8 byte hashes (rather than 2^{83} 16 byte hashes).

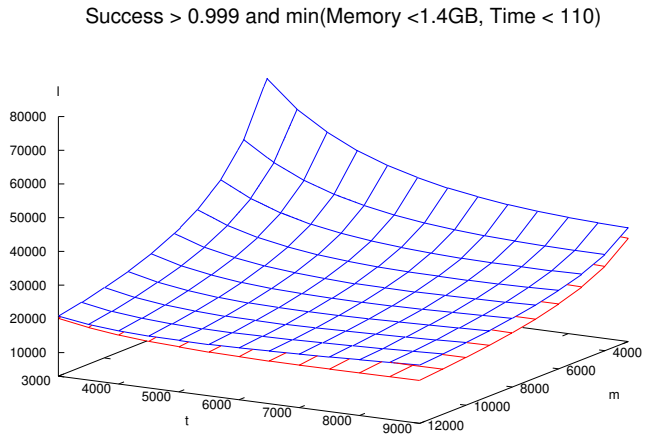

Fig. 3. Comparison of the success rate of classical tables and rainbow tables. The upper surface represents the constraint of 99.9% success with classical tables, the lower surface is the same constraint for rainbow tables. For rainbow tables the scale has been adjusted to allow a direct comparison of both types of tables $m \rightarrow \frac{m'}{t}, \ell \rightarrow \frac{\ell'}{t}$

Based on Figure 1 we have chosen the parameters for classic tables to be $t_c = 4666, m_c = 8192$ and for rainbow tables to be $t_r = 4666, m_r = t_c \times m_c = 38'223'872$. We have generated 4666 classic tables and one rainbow table and measured their success rate by cracking 500 random passwords on a standard workstation (P4 1.5GHz, 500MB RAM). The results are given in the table below:

	classic with DP	rainbow
t, m, ℓ	4666, 8192, 4666	4666, 38'223'872, 1
predicted coverage	75.5%	77.5%
measured coverage	75.8%	78.8%

Table 1. Measured coverage for classic tables with distinguished points and for rainbow tables, after cracking of 500 password hashes

This experiment clearly shows that rainbow tables can achieve the same success rate with the same amount of data as classical tables. Knowing this, it is now interesting to compare the cryptanalysis time of both methods since rainbow tables should be twice as fast. In Table 2 we compare the mean cryptanalysis time, the mean number of hash operations per cryptanalysis and the mean number of false alarms per cryptanalysis.

What we see from table 2 is that our method is actually about 7 times faster than the original method. Indeed, each cryptanalysis incurs an average of 9.3M hash calculations with the improved method whereas the original method incurs 67.2M calculations. A factor of two is explained by the structure of the tables. The remaining speed-up is caused by the fact that there are more false alarms with distinguished points (2.8 times more in average) and that these false alarms generate more work. Both effects are due to the fact that with distinguished points, the length of the chains is not constant.

4.1 The importance of being constant

Fatal attraction: Variations in chain length introduce variations in merge probability. Within a given set of chains (e.g. one table) the longer chains will have more chances to merge with other chains than the short ones. Thus the merges will create larger trees of longer chains and smaller trees of shorter chains. This has a doubly negative effect when false alarms occur. False alarm will more probably happen with large trees because there are more possibilities to merge into a large tree than into a small one. A single merge into a large tree creates more false alarms since the tree contains more chains and all chains have to be generated to confirm the false alarm. Thus false alarms will not only tend to happen with longer chains, they will also tend to happen in larger sets.

Larger overhead: Additionally to the attraction effect of longer chains, the number of calculations needed to confirm a false alarm on a variable length chains is larger than with constant length chains. When the length of a chain is not known the whole chain has to be regenerated to confirm the false alarm. With constant length chains we can count the number of calculations done to reach the end of a chain and then know exactly at what position to expect the key. We thus only have to generate a fraction of a chain to confirm the false alarm. Moreover, with rainbow chains, false alarms will occur more often when we look at the longer chains (i.e. starting at the columns more to the left of a table). Fortunately, this is also where the part of the chain that has to be generated to confirm the false alarms is the shortest.

Both these effects can be seen in Table 2 by looking at the number of endpoints found, the number of false alarms and the number of calculations per false alarm, in case of failure. With distinguished points each matching point generates about 4 false alarms and the mean length of the chains generated is about 9600. With rainbow chains there are only about 2.5 false alarms per endpoint found and only 1500 keys generated per false alarm.

The fact that longer chains yield more merges has been noted in [7] without mentioning that it increases the probability and overhead of false alarms. As a result, the authors propose to only use chains which are within a certain range of length. This reduces the problems due to the variation of length but it also reduces the coverage that can be achieved with one reduction function and increases the precalculation effort.

	classic with DP	rainbow	ratio
t, m, ℓ	4666, 8192, 4666	4666, 38'223'872, 1	1
	mean cryptanalysis time		
to success	68.9s	9.37s	7.4
to failure	181.0s	26.0s	7.0
average	96.1s	12.9s	7.4
	mean nbr of hash calculations		
to success	48.3M	6.77M	7.1
to failure	126M	18.9M	6.7
average	67.2M	9.34M	7.2
	mean nbr of searches		
to success	1779	2136	0.83
to failure	4666	4666	1
average	2477	2673	0.93
	mean nbr of matching endpoints found		
to success	1034	620	1.7
to failure	2713	2020	1.3
average	1440	917	1.6
	mean nbr of false alarms		
to success	4157	1492	2.8
to failure	10913	5166	2.1
average	5792	2271	2.6
	mean nbr of hash calculations per false alarms		
to success	9622	3030	3.2
to failure	9557	1551	6.2
average	9607	2540	3.8

Table 2. statistics for classic tables with distinguished points and for rainbow tables

4.2 Increasing the gain even further

We have calculated the expected gain over classical tables by considering the worst case where a key has to be searched in all columns of a rainbow table and without counting the false alarms. While a rainbow table is searched from the amount of calculation increases quadraticly from 1 to $\frac{t^2-1}{2}$, whereas in classical tables it increases linearly to t^2. If the key is found early, the gain may thus be much higher (up to a factor of t). This additional gain is partly set off by the fact that in rainbow tables, false alarms that occur in the beginning of the search, even if rarer, are the ones that generate the most overhead. Still, it should be possible to construct a (possibly pathological) case where rainbow tables have an arbitrary large gain over classical tables. One way of doing it is to require a success rate very close to 100% and a large t. The examples in the litterature often use a success rate of up to 80% with $N^{1/3}$ tables of order of $N^{1/3}$ chains of $N^{1/3}$ points. Such a configuration can be replaced with a single rainbow table of order of $N^{2/3}$ rows of $N^{1/3}$ keys. For some applications a success rate of 80% may be sufficient, especially if there are several samples of ciphertext available and we

need to recover just any key. In our example of password recovery we are often interested in only one particular password (e.g. the administrator's password). In that case we would rather have a near perfect success rate. High success rates lead to configurations where the number of tables is several times larger than the length of the chains. Thus we end up having several rainbow tables (5 in our example). Using a high success rate yields a case were we typically will find the key early and we only rarely have to search all rows of all tables. To benefit from this fact we have to make sure that we do not search the five rainbow tables sequentially but that we first look up the last column of each table and then only move to the second last column of each table. Using this procedure we reach a gain of 12 when using five tables to reach 99.9% success rate compared to the gain of 7 we had with a single table and 78% success rate. More details are given in the next section.

4.3 Cracking Windows passwords in seconds

After having noticed that rainbow chains perform much better than classical ones, we have created a larger set of tables to achieve our goal of 99.9% success rate. The measurements on the first table show that we would need 4.45 tables of 38223872 lines and 4666 columns. We have chosen to generate 5 tables of $35'000'000$ lines in order to have an integer number of tables and to respect the memory constraint of $1.4GB$. On the other hand we have generated $23'330$ tables of 4666 columns and 7501 lines. The results are given in Table 3. We have cracked 500 passwords, with 100% success in both cases.

	classic with DP	rainbow	ratio	rainbow sequential	ratio
t, m, ℓ	4666, 7501, 23330	4666, 35M, 5	1	4666, 35M, 5	1
cryptanalysis time	101.4s	66.3	1.5	13.6s	**7.5**
hash calculations	90.3M	7.4M	**12**	11.8M	7.6
false alarms (fa)	7598	1311	5.8	2773	2.7
hashes per fa	9568	4321	2.2	3080	3.1
effort spent on fa	80%	76%	1.1	72%	1.1
success rate	100%	100%	1	100%	1

Table 3. Cryptanalysis statistics with a set of tables yielding a success rate of 99.9%. From the middle column we see that rainbow tables need 12 times less calculations. The gain in cryptanalysis time is only 1.5 times better due to disk accesses. On a workstation with 500MB of RAM a better gain in time (7.5) can be achieved by restricting the search to one rainbow table at a time (rainbow sequential).

From table 3 we see that 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. This is because we have to randomly access 1.4GB of data on a workstation that has 500MB of RAM. In the previous measurements with a

single table, the table would stay in the filesystem cache, which is not possible with five tables. Instead of upgrading the workstation to 1.5GB of RAM we chose to implement an approach where we search in each rainbow table sequentially. This allows us to illustrate the discussion from the end of the previous section. When we search the key in all tables simultaneously rather than sequentially, we work with shorter chains and thus generate less work (7.4M operations rather than 11.8M). Shorter chains also mean that we have less false alarms (1311 per key cracked, rather than 2773). But short chains also mean that calculations needed to confirm a false alarm are higher (4321 against 3080). It is interesting to note that in all cases, the calculations due to false alarms make about 75% of the cryptanalysis effort.

Looking at the generic parameters of the trade-off we also note that the precalculation of the tables has needed an effort about 10 times higher than calculating a full dictionary. The large effort is due to the probabilistic nature of the method and it could be reduced to three times a full dictionary if we would accept 90% success rate rather that than 99.9%.

5 An outlook at perfect tables

Rainbow tables and classic tables with distinguished points both have the property that merging chains can be detected because of their identical endpoints. Since the tables have to be sorted by endpoint anyway, it seems very promising to create perfect tables by removing all chains that merge with chains that are already in the table. In the case of distinguished points we can even choose to retain the longest chain of a set of merging chains to maximise the coverage of the table. The success rate of rainbow tables and tables with distinguished points are easy to calculate, at least if we assume that chains with distinguished points have a average length of t. In that case it is straight forward to see that a rainbow table of size $mt \times t$ has the same success rate than t tables of size $m \times t$. Indeed, in the former case we have t rows of mt distinct keys where in the latter case we have t tables containing mt distinct keys each.

Ideally we would want to construct a single perfect table that covers the complete domain of N keys. The challenge about perfect tables is to predict how many non-merging chains of length t it is possible to generate. For rainbow chains this can be calculated in the same way as we calculate the success rate for non-perfect tables. Since we evaluate the number of distinct points in each column of the table, we need only look at the number of distinct points in the last column to know how many distinct chains there will be.

$$\hat{P}_{table} = 1 - e^{-t\frac{m_t}{N}} \quad \text{where} \quad m_1 = N \quad \text{and} \quad m_{n+1} = N\left(1 - e^{-\frac{m_n}{N}}\right) \tag{4}$$

For chains delimited by distinguished points, this calculation is far more complex. Because of the fatal attraction described above, the longer chains will be merged into large trees. Thus when eliminating merging chains we will eliminate

more longer chains than shorter ones. A single experiment with 16 million chains of length 4666 shows that after elimination of all merges (by keeping the longest chain), only 2% of the chains remain and their average length has decreased from 4666 to 386! To keep an average length of 4666 we have to eliminate 96% of the remaining chains to retain only the longest 4% (14060) of them.

The precalculation effort involved in generating maximum size perfect tables is prohibitive (Nt). To be implementable a solution would use a set of tables which are smaller than the largest possible perfect tables.

More advanced analysis of perfect tables is the focus of our current effort. We conjecture that because of the limited number of available non-merging chains, it might actually be more efficient to use near-perfect tables.

6 Conclusions

We have introduced a new way of generating precomputed data in Hellman's original cryptanalytic time-memory trade-off. Our optimisation has the same property as the use of distinguished points, namely that it reduces the number of table look-ups by a factor which is equal to the length of the chains. For an equivalent success rate our method reduces the number of calculations needed for cryptanalysis by a factor of two against the original method and by an even more important factor (12 in our experiment) against distinguished points. We have shown that the reason for this extra gain is the variable length of chains that are delimited by distinguished points which results in more false alarms and more overhead per false alarm. We conjecture that with different parameters (e.g. a higher success rate) the gain could be even much larger than the factor of 12 found in our experiment. These facts make our method a very attractive replacement for the original method improved with distinguished points.

The fact that our method yields chains that have a constant length also greatly simplifies the analysis of the method as compared to variable length chains using distinguished points. It also avoids the extra precalculation effort which occurs when variable length chains have to be discarded because they have an inappropriate length or contain a loop. Constant length could even prove to be advantageous for hardware implementations.

Finally our experiment has demonstrated that the time-memory trade-off allows anybody owning a modern personal computer to break cryptographic systems which were believed to be secure when implemented years ago and which are still in use today. This goes to demonstrate the importance of phasing out old cryptographic systems when better systems exist to replace them. In particular, since memory has the same importance as processing speed for this type of attack, typical workstations benefit doubly from the progress of technology.

Acknowledgements

The author wishes to thank Maxime Mueller for implementing a first version of the experiment.

References

1. J. Borst, B. Preneel, and J. Vandewalle. On time-memory tradeoff between exhaustive key search and table precomputation. In P. H. N. de With and M. van der Schaar-Mitrea, editors, *19th Symp. on Information Theory in the Benelux*, pages 111–118, Veldhoven (NL), 28-29 1998. Werkgemeenschap Informatie- en Communicatietheorie, Enschede (NL).
2. D.E. Denning. *Cryptography and Data Security*, page 100. Addison-Wesley, 1982.
3. Amos Fiat and Moni Naor. Rigorous time/space tradeoffs for inverting functions. In *STOC 1991*, pages 534–541, 1991.
4. M. E. Hellman. A cryptanalytic time-memory trade off. *IEEE Transactions on Information Theory*, IT-26:401–406, 1980.
5. Kim and Matsumoto. Achieving higher success probability in time-memory trade-off cryptanalysis without increasing memory size. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, 1999.
6. Koji KUSUDA and Tsutomu MATSUMOTO. Optimization of time-memory trade-off cryptanalysis and its application to DES, FEAL-32, and skipjack. *IEICE Transactions on Fundamentals*, E79-A(1):35–48, January 1996.
7. F.X. Standaert, G. Rouvroy, J.J. Quisquater, and J.D. Legat. A time-memory tradeoff using distinguished points: New analysis & FPGA results. In *proceedings of CHES 2002*, pages 596–611. Springer Verlag, 2002.

7 Appendix

The success rate of a single rainbow table can be calculated by looking at each column of the table and treating it as a classical occupancy problem. 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:

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

Each column i has m_i distinct keys. The success rate of the table is thus:

$$\begin{aligned} P &= 1 - \prod_{i=1}^{t}(1 - \frac{m_i}{N}) \\ &\text{where} \\ m_1 = m \quad &, \quad m_{n+1} = N\left(1 - e^{-\frac{m_n}{N}}\right) \end{aligned}$$

The result is not in a closed form and has to be calculated numerically. This is no disadvantage against the success rate of classical tables since the large number of terms in the sum of that equation requires a numerical interpolation.

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.

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

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

$$\hat{P} = 1 - (1 - \frac{m_t}{N})^t \approx 1 - e^{-t\frac{m_t}{N}}$$

Note that the effort to build such a table is Nt.