Lecture 29-04-2021

Filippo Baudanza

May 8, 2021

1 Hellman's and Rainbow tables

1.1 Introduction

***Abstract*—A probabilistic method is presented which cryptanalyzes any N key cryptosystem in $N^{2/3}$ operations with $N^{2/3}$ words of memory (average values) after a precomputation which requires N operations. If the precomputation can be performed in a reasonable time period (e.g., several years), the additional computation required to recover each key compares very favorably with the N operations required by an exhaustive search and the N words of memory required by table lookup. When applied to the Data Encryption Standard (DES) used in block mode, it indicates that solutions should cost between $1 and $100 each. The method works in a chosen plaintext attack and, if cipher block chaining is not used, can also be used in a ciphertext-only attack.**

I. Introduction

MANY SEARCHING tasks, such as the knapsack [1] and discrete logarithm problems [2], allow time–memory trade-offs. That is, if there are N possible solutions to search over, the time–memory trade-off allows the solution to be found in T operations (time) with M words of memory, provided the time–memory product TM equals N. (Often the product is of the form $cN \log_2 N$, but for simplicity we neglect logarithmic and constant factors.)

The cryptanalyst enciphers some fixed plaintext P0 under each N = |K| keys. This produce a table $(Enc_k(P_0), k)$ which is sorted by $Enc_k(P_0)$ and stored. When a user chooses a new key k, he is forced (CPA-attack) to provide $Enc_k(P0)$ to the cryptanalyst. Since the look up table is sorted the cryptanalyst running time to find k is O(log(N)).

The time–memory trade-off is best understood by considering a specific cryptosystem, such as the DES. It operates on a 64-bit plaintext block P to produce a 64-bit ciphertext block C under the action of a 56-bit key:

$$C = S_K(P). \tag{2}$$

Letting P_0 be a fixed plaintext block, define

$$f(K) = R[S_K(P_0)], \tag{3}$$

where R is some simple reduction from 64 to 56 bits, such as dropping the last 8 bits of the ciphertext. Fig. 1 depicts the construction of f.

Computing $f(K)$ is almost as simple as enciphering, but computing K from $f(K)$ is equivalent to cryptanalysis. If the cryptosystem is secure, f is therefore a one-way function [3]. The time–memory trade-off described in this paper applies to inverting any one-way function, not just those derived from cryptosystems.

1.2 Hellman's tables

Hellman's and Rainbow tables

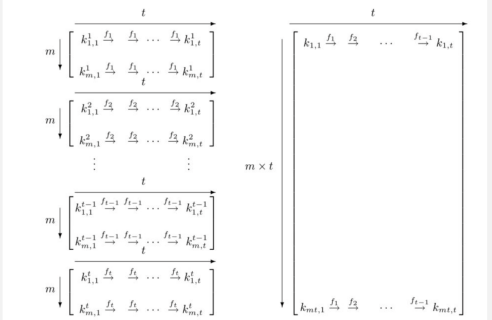

Fig. 2. t classic tables of size $m \times t$ on the left and one rainbow table of size $mt \times t$ on the right. In both cases merges can occur within a group of mt keys and a collision can occur with the remaining $m(t-1)$ keys. It takes half as many operations to look up a key in a rainbow table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

Here are Hellman's two main ideas: First one: In one classic Hellman's table m keys k1, · · · , km are chosen uniform and random from the key space K. To reduce memory just the first and last column are stored before been sorted by the last column. Now assume that someone chooses a key k and the cryptanalyst intercepts Y1 = f(k). So he checks if Y1 is in the table. If yes then he knows the key is in the next to the last column of a given arrow hence he recover the key with O(t) operations. Otherwise he compute Y2 = f(Y1) and checks if it is the

last column. Notice that to look up a key t search operations are necessary. If all m $\times$ t elements are different and random the success probability is $\frac{mt}{N}$. Second one: t classical tables are generated using random "reductions" R. Namely, t random bijections R are chosen and fR := R $\circ$ f is used as above keeping in mind that once intercepted f(k) we have to check against Y1 = R(f(k)). Now we have success probability mt2 . Notice that to look up a key t2 = t $\times$ t search operations are N necessary.

1.3 Rainbow tables

Rainbow tables effectively solve the problem of collisions with ordinary hash chains by replacing the single reduction function R with a sequence of related reduction functions R_1 through R_k. In this way, for two chains to collide and merge they must hit the same value on the same iteration: consequently, the final values in these chain will be identical. A final postprocessing pass can sort the chains in the table and remove any "duplicate" chains that have the same final values as other chains. New chains are then generated to fill out the table. These chains are not collision-free (they may overlap briefly) but they will not merge, drastically reducing the overall number of collisions. Using sequences of reduction functions changes how lookup is done: because the hash value of interest may be found at any location in the chain, it's necessary to generate k different chains. The first chain assumes the hash value is in the last hash position and just applies R_k; the next chain assumes the hash value is in the second-to-last hash position and applies R_k1, then H, then R_k; and so on until the last chain, which applies all the reduction functions, alternating with H. This creates a new way of producing a false alarm: if we "guess" the position of the hash value wrong, we may needlessly evaluate a chain. Although rainbow tables have to follow more chains, they make up for this by having fewer tables: simple hash chain tables cannot grow beyond a certain size without rapidly becoming inefficient due to merging chains; to deal with this, they maintain multiple tables, and each lookup must search through each table. Rainbow tables can achieve similar performance with tables that are k times larger, allowing them to perform a factor of k fewer lookups.

2 Salt

One of the earlier applications of cryptographic hash functions was the storage of passwords for user authentication in computer systems. With this method, a password is hashed after its input and is compared to the stored (hashed) reference password. People realized early that it is sufficient to only store the hashed versions of the passwords. A salt is a random value appended to the password before hashing.

> Different users, same password. Different salts, different hashes. If someone looked at the full list of password hashes, no one would be able to tell that Alice and Bob both use the same password. Each **unique salt** extends the password `farm1990M00` and transforms it into a **unique password**.

In practice, we store the salt in cleartext along with the hash in our database. We would store the salt `f1nd1ngn3m0`, the hash `07dbb6e6832da0841dd79701200e4b179f1a94a7b3dd26f612817f3c03117434`, and the username together so that when the user logs in, we can lookup the username, append the salt to the provided password, hash it, and then verify if the stored hash matches the computed hash.

Now we can see why it is very important that each input is salted with unique random data. When the salt is unique for each hash, we inconvenience the attacker by now having to compute a rainbow table for each user hash. This creates a big bottleneck for the attacker. Ideally, we want the salt to be truly random and unpredictable to bring the attacker to a halt.