Hellman attacks - track3

Introduction

Let's say we have a random, cryptographic function f:{0,1}^k -> {0,1}^k (i.e. taking as input a k-length bits number and producing a k-length bits number). We need to "invert" a y: find **all** inputs x such as f(x) = y.

It is possible to obtain a tradeoff between time and memory, as described in this rainbow table paper (https://lasec.epfl.ch/pub/lasec/doc/Oech03.pdf). We want to answer an invert y operation in O(n^(⅔)) time, but storing only O(n^(⅔)) "checkpoints" in memory (instead of the usual O(n) stored memory).

Rainbow tables

This paragraph describes the main ideas behind the paper. Say we want to store only O(n^(⅔)) f outputs. In order to cover all O(n) outputs, the stored checkpoints should be enough to recover everything by pure calculation, so each of the stored values should give us O(n^(⅓)) distinct extra outputs in ideal case (O(n^(⅔) stored points * O(n^(⅓)) covered values by each stored point gives us O(n) covered points).

Let's say we want to store only one value. How can we cover cover O(n^(⅓)) distinct values? Say the starting point is x0. Let x1 = f(x0), x2 = f(x1) = f(f(x0)), .., xk = f(x[k-1]) = f^k(x0). By f^k(x) we mean applying function f k times, each time taking as an argument the previous result and starting with x (function composition). Say the value y appears in this array. Let xi = y. Then, we know x[i-1] is the answer we are looking for, as xi was defined as f(x[i-1]). Note that only x0 needs to be stored in memory in this case, the other ones can be calculated by incrementally applying function f to the previous result.

This leads to the first naive idea: store x1, x2, …, xm as the beginning of each m array, choose m = n^(⅔), and pick the length of each array as L=n^(⅓). Given a challenge y, we need to find in which of the m chains it appears, and in which position(s) it appears in each chain. This will give us the solution set, by looking at the element before each position in each chain.

Forward evaluating each chain, from the beginning to the end is obviously not possible, as it leads to O(n) time needed to invert a challenge y (there are O(n^(⅔)) chains, each needing O(n^(⅓)) time to be fully evaluated). We need to somehow evaluate all chains "in parallel". To fix this, we'll also store in memory the end of the chains: y1, y2, …, ym, where yi is the result after evaluating xi L times (yi = f^L(xi)).

Now, say we want to find if the value y appears at a position p in any of the chains. This means xp = y for a chain. In turn, this means x[p+1] = f(y), x[p+2] = f(f(y)), …, x[L] = f^(L-p)(y). So, given that y appears at position p in a chain, we know the end of the chain must be y' = f(f(f(...(y))) (applied L-p times). If y' doesn't appear at all in array y1, y2, …, ym we know for sure that y doesn't appear at position p in any chain! (otherwise, we'd surely find y' as an end of a chain). If we find y' in the end of chains vector, we can retrieve which chain(s) contain it. Say y3 = y4 = y5 = y'. We can retrieve x3, x4, x5. We then check if f^p(x3) = y3 (is y3 really appearing in position p on chain started by x3?). If so, we know we found a solution, which is f^(p-1)(x3). We do similarly for x4 and x5.

To quickly find the endpoints where a value can appear, we sort the pairs (start_point, end_point) by end_point. Now, the positions where y' appears form a contiguous subarray. This means, in the vector, all positions where y' appears are between a low and a high position. (i.e. low <= pos <= high, for some calculated low and high, such as end_point[pos] = y'). Given that the vector of end_points is sorted in ascending order, we can binary search the low position as well as high position. Then, we iterate over the range [low, high]. All the chains will end in y', and we can simply retrieve the starting point of the chain by using the start_point vector.

Iterating over p=1,2,..., L gives us the full set of solutions. Finding the end of chain value takes O(L). Also checking all the candidates chains and retrieving the correct solution takes O(L). So we have O(L) candidate positions, each taking O(L) time to evaluate, giving us O(L^2) time to answer a query y, or O(n^(⅔)), given that L=n^(⅓).

There is a main problem with this approach: say on two chains, the value x appears. Those chains will have, by that point, all values overlapping: both of them will contain the values f(x), f(f(x)), ... This is clearly bad, as we stated each chain should cover O(n^(⅓)) distinct values in order for this to work (i.e. to cover all possible O(n) inputs). After we find x again on the second chain, the remaining part of the chain is simply wasted, as all those inputs were already covered by the first chain already.

When an x value appears on two chains, we would want the next elements on the two chains to be different. We'll refine the idea in the following way: we previously used x[i] = f(x[i-1]). Now, we'll use x[i] = f(i, x[i-1]), in other words, the next value on the chain depends not only on the previous value, but also on the position on the chain of the previous value.

For example, we have x0, y0 at the beginning of two chains. Now, we do x1 = f(x0), then x1 = P1(x1), similarly y1 = f(y0), then y1 = P1(y1). The function P1 takes an input and simply shuffles its bits. Then, we do x2 = f(x1), x2 = P2(x1), y2 = f(y1), y2 = P2(y2). P2 function takes an input an shuffles its bits, but in a different way than P1 did. We continue to use functions P3, P4, …, PL, all shuffling bits in different ways. The chains look like this:

x0 -> x1=P1(f(x0)) -> x2 = P2(f(x1)) -> x3 = P3(f(x2)) -> …
y0 -> y1=P1(f(y0)) -> y2 = P2(f(y1)) -> y3 = P3(f(y2)) -> …

z0 -> z1=P1(f(z0)) -> z2 = P2(f(z1)) -> z3 = P3(f(z2)) -> ...

In our implementation, P functions are stored as permutations. For example, take permutation {3, 0, 1, 2} and x = 10 = 1010 in binary. Then, P(10) = 0101 = 5. Permutations are stored explicitly, each permutation storing k integers, requiring a total of k*2^(k/3) additional memory. This can be avoided by choosing smarter P functions than permutations (whilst still keeping good randomness of the output). The only requirement of P functions is to produce random outputs and each two to be as different as possible, given the same inputs. The permutation of the bits of input is a natural choice here. Given that the algorithm already uses O(2^(2k/3)) memory, we consider the O(k*2^(k/3)) factor insignificant in overall memory complexity.

This solves our previous problem: say the same value x appears in positions p and q in two chains. Previously, this would mean the next values will coincide (i.e. be equal to f(x)). Now, one chain will have the value P(p+1)(f(x)), the other will have the value P(q+1)(f(x)). Since our assumption is P functions produce outputs as different as possible two by two, we can assume the outputs will be different, given the same input (f(x)). The only chance the outputs would be equal is if p = q, but this happens only with 1/L probability (as we consider the chain construction to be random). So the probability to get two identical xs on two chains, and the chains to merge (and the second chain to store useless information) is only 1/L now.

Note that the same algorithm still works, but now when we check if a y value appears at position p, we'll need to search the value Pp(y) instead of y. This is because, if f(x[p-1]) = y, then the element on chain at position p will not be f(x[p-1]) anymore, but Pp(f(x[p-1])). So checking that f(x[p-1]) = y is equivalent to checking that Pp(f(x[p-1])) = Pp(y), which in turn means Pp(y) appears at position p in a chain (then, x[p-1] is the solution we're looking for).

Inverting Plotter f1

Recall that f1 used in plotter is actually outputting k+5 bits instead of k (i.e. f1:{0,1}^k -> {0,1}^(k+5)). In order to keep the function from k to k bits, we'll drop the last 5 bits of every f call in our tables. Now say we're inverting an y. We'll drop the last 5 bits of y as well and invert that value. We'll get a list of x candidates. We now need to call the real f1 function for each candidate to check which of them are the real inverses.

Getting perfect accuracy

The tables will never be able to invert 100% of y values. They can have high accuracy (lots of y values to appear somewhere in the chains) but it's unlikely all y values will appear. In order to answer every challenge, we need 100% accuracy of f1 inversions. We'll store in memory the x values that never appear in the table.

We'll use a temporary file (similarly to the one plotter uses) in order to find all x values that do not appear in the chains. To do that, firstly we write in the temporary file all x values that appear: we iterate over each chain, and write on the disk file every value we encounter. To iterate over each chain, simply start with its start value (stored in memory), apply the function L-1 times, and each time write into the temporary file the value encountered (we'll be storing start_chain, f(start_chain), …, f^(L-1)(start_chain), for each start_chain from memory).

Now, let's sort on disk all the values stored and iterate over them in the sorted order. The values that appear will look like in this example: x=0,0,1,1,3,4,4,4,7,7 and so on. From this example, we can conclude that x=2, x=5 and x=6 were never produced by the Rainbow table. In other words, the "gaps" between the sorted values represent what we are looking for.

The algorithm goes like this: iterate the temporary file after the sorting, and let prev_x be the previously read value, and cur_x the current read value. The gap between prev_x and cur_x is given by the set {prev_x + 1, prev_x + 2, …, cur_x - 1}. So, if prev_x + 1 < cur_x, we store in the extra_metadata memory all the values from {prev_x + 1, prev_x + 2, …, cur_x - 1}, since we know they'll never be produced by the table.

Now, we'll store the pairs {f1(extra_metadata[i]), metadata[i]} in the memory and sort them by the first field. When inverting a value y, beside querying the table, we'll also have to look into the extra storage memory: since the extra storage pairs are sorted by the first field, we can binary search y to find all the positions where it appears (once again, it'll be a contiguous subsequence thanks to sorting). Once we have all the positions of the pairs, we can take the second field of each pair and add them to the inversions set for y (together with the ones obtained by the table itself). This ensures all inverses of y can be found.

Attacking plotter challenges

Recall a proof to a challenge are 64 k-bit values: x1, x2, …, x64 such as f1(x1) matches f1(x2), f1(x3) matches f1(x4), …, f2(x1, x2) matches f2(x3, x4), …, f3(x1, x2, x3, x4) matches f3 (x5, x6, x7, x8), …, f7(x1, x2, …, x64) = challenge.

In the final file, table 1 stores x1x2 (such as they match with respect to f1), table2 stores x1x2x3x4 (such as they match with respect to both f1 and f2) by the form of 2 pointers to table 1, one for x1x2 and one for x3x4, table3 stores x1x2..x8 (by 2 pointers to table 2, one for x1x2x3x4 and one for x5x6x7x8) and so on.

We are going to drop one table from the final file, the first one. Table 2 now stores x1 and x3 (note that x2 and x4 are "missing" for now, the values such as f1(x1) matches f1(x2) and f1(x3) matches f1(x4), f2(x1, x2) matches f2(x3, x4). Table 3 stores two pointers to the entries of table 2 and so on. Table 1 will be almost empty, so we'll save space in the final file (what's stored there will be discussed later).

Doing the proving process and retrieving data from the final file, we'll get x1, x3, x5, …, x63. We need now to "guess" x2, x4, x6, …, x64.

We start by having candidate pairs: for (x1, x3), we'll have a vector of candidate pairs (x2, x4), similarly, for (x5, x7), we'll have a vector of candidate pairs (x6, x8) and so on. Hence, we have a total of 16 vectors of pairs. Getting full proof becomes now brute-forcing over all candidate pairs, until we find one full proof that respects all the conditions. This is done in TryAllProofs function from prover, recursively picking all pairs from all vectors, until we find one proof that's good.

In order to find all candidates (x2, x4) for a pair (x1, x3), we'll get a list of all x2 that match x1 with respect to f1 and a list of all x4 that match with x3 with respect to f1. Then, try two-by-two all x2 and x4 from the lists, and check if f2(x1, x2) matches with f2(x3, x4). If so, store (x2, x4) as a candidate for (x1, x3). This is implemented in InvertF2 from the prover.

Finally, given a x1, we need to find all x2 such as f1(x1) matches with f1(x2). Let y = f1(x1). We know all y2 that match with y, using the matching-shifts formula. For each y2, we invert them using the Hellman attacks, and store all x2 we found such as f1(x2) = y2. This is implemented in InvertF1.

Summing up, Hellman attacks are used to find all x2 such as f1(x1) matches f1(x2) (firstly guessing the matching ys with f1(x1), then reverting them with Hellman attacks). Then, in order to invert a pair (x1, x3), we find all x2 and x4 lists (using InvertF1), try two-by-two all values, and see which f2(x1, x2) matches with f2(x3, x4). Finally, inverting all pairs (x1, x3), (x5, x7), …, (x61, x63), we brute force over all candidates to find a good matching proof.

There is no need in this approach to store anything in table 1. However, we'll store extra storage memory (discussed above), in order to sync the Hellman tables from the plotter with the one from the prover. The extra storage is calculated in the plotter (using the same temporary file as in the plotting process), then the results are copied in the final file table 1, from where the prover can load them. The reason we implement extra storage in the plotting process is that, once the plotting is done, the algorithm not to use any more disk space than the one from the final file (doing extra storage in prover, for example, would require creating a temporary file, so we'd need extra memory in proving other than the final file, which is forbidden). The temporary file used for extra storage phrase is the same as the one used for plotting, so no memory waste is done here (the file will just be overwritten in the plotting phrase, after extra storage phrase is done).

Limitations of the approach / possible improvements

Both tables and extra storage are stored in the memory. Using 5 tables, each containing n^(⅔) chains, will result in approximately 98% accuracy. The remaining 2% of table 1 will be stored in memory. For big k, those 2% elements won't fit in memory, so a better implementation must

deal with them on disk (instead of loading them in memory, just use them from table1 final file). Moreover, for even bigger k, not even the table begins/ends will fit into the memory (this should be a problem for k around 40-45).