Reverse engineering of one-way encryption function

Nikolay Raychev

Abstract - in this research is described an approach for reverse engineering of a hash function. Despite that the proposed method may not be the best, it still works and it deserves to be examined. The approach does a bloom filter containing states that may reach the end by adding a suffix with some length. Then it iterates the prefixes of the extra length and notes each one that matches the filter. As soon as it reaches ten thousand corresponding prefixes or ends, it tries to break recursively the difference from the states, reached by comparing prefixes against the end state. If it finds a way to break the difference, the correct prefix forms a pair with the solution of the difference, in order to form a complete solution. Otherwise, it continues, until the prefixes run out.

Key words: Quantum computing, diffraction, simulator, operators, gates

1. INTRODUCTION

The hashing is a process of converting a sequence of symbols into another value that corresponds to the original one by using one-way functions and it is practically impossible the original value to be converted by another algorithm only from the hash value.

Some of the most important properties of the hashing algorithms:

• The resulting hash should be as random as possible i.e. so that any assumptions and conclusions for the original text can not be made based on the resulting hash.

• The hashing function should have a high entropy i.e. the chance for collision (equal hash at two different initial texts) should be minimum (ideally zero).

• They must be slow. If the cracker knows which hashing algorithm is used it can generate a rainbow table i.e. to make to itself matches of hashes and their initial values. A slower algorithm would have slowed down repeatedly the generation of such a table.

The hash function encodes plain text with variable length into a hash value with fixed length, the hashing is often used for signing the data or upon their authentication. As it is known, the secure hash function must comply with several requirements: it must be one-way, to be a secure protection against birthday and against meet-in-the-middle attacks. A number of publications from the last ten years prove that these widely used hash functions such as MD5 or SHA-1 are no longer secure. In this way, new hash functions must be examined, in order to meet the practical needs of applications for greater cryptographic security.

The one-way encryption functions can be used for protection of passwords. The idea is that someone with access to the hash, can not determine the corresponding password, but he can use it, to recognize the password when he receives it. This is especially useful in cases when hackers have access to a source code or data. For protection of brute force attacks are used techniques, by which is slowed down the generation of the final look of our password. One of the frequently used is key stretching or multiple hashing, whose purpose is to complicate and delay the algorithm for hashing, so that the hardware of the cracker may not be able to handle it for a short period of time. The most simple method for key stretching is the application of a hash function on the result from it for example 1,000 times. Another approach is to use various heavier algorithms when possible such that are using 64 bit operations and are 'more difficult' for the modern video cards (bcrypt, scrypt, sha-512). Here, however, should be selected the limit, because this technique will slow down both the cracker and our server during registration/login of normal users i.e. we will solve one problem, but at the same time will be exposed to another vulnerability - DDOS attacks.

In this article is described an approach for reverse engineering of a hash function. Despite that the

proposed method may not be the best, it still works and it deserves to be examined.

2. THE APPROACH

One-way encryption function

To be clearly understandable by the readers the one-way encryption function is rewritten to C#:

```
static hashHashTuple<Int32, Int32> Hash(string
text) {
   var
dictCharSet="abcdefghijklmnopqrstuvwxyzABCD
EFGHIJKLMNOPQRSTUVWXYZ0123456789`~!@#
$%^&*()_+-=|[];',.{}:<>? ";
   Int32 a = 0;
   Int32 b = 0;
   foreach (var letter in text) {
      var e = dictCharSet.IndexOf(letter);
      if (e == -1) e = dictCharSet.Length + 1;
      for (var i = 0; i < 17; i++) {
         a = a *-6 + b + 0x74FA - e;
         b = b / 3 + a + 0x81BE - e;
      }
   }
   return HashTuple.Create(a, b);
}
```

As can be seen, the state of the one-way encryption function consists of 32-bit signed integers (a, b), which start from 0. The input is a sequence of symbols, made up of 93 possibilities. Each symbol from the input is added into state on a progression of 17 rotations and when the last symbol has been added, the result is the final state (a, b).

It should be noted that the addition and multiplication are not verified (e.g. Int32.MaxValue+1 = Int32.MinValue, Int32.MaxValue*2 = -2) and the division is rounded towards 0 (e.g. -4/3 = -1, 7/3 = 2).

In addition to the one-way encryption function there is also a translated code that verify whether the combination of username/password is valid:

```
static bool CheckHashVerify(string
username, string password) {
   var
CheckExpectedResultPassHash  = HashTuple.Creat
e(-0x20741256, -0x4A579222);
   var CheckExpectedResultNameHashes = new[] {
      HashTuple.Create(-0x52BEB283, -0x733C9599),
      HashTuple.Create(0x605D4A4F, 0x7EDDB1E5)
,
      HashTuple.Create(0x3D10F092, 0x60084719)
   };

   var ResultPassHash = Hash(password);
   var ResultNameHash = Hash(username);
   return password.StartsWith("<+")
      &&
ResultPassHash.Equals(CheckExpectedResultPass
Hash )
      &&
CheckExpectedResultNameHashes.Contains(Resul
tNameHash);
}
```

As can be seen, both the valid user names and the valid password are protected by hashing. Also, the first two characters of the password are included in the code.

Although it may seem meaningless to give away some of the symbols of the password, in fact it is a good idea in the given context. The prefix is used as a filter for the event that triggers the hashing, in order to avoid the hashing of each message, said by someone. The filter allows not to be shared secretly all messages. The corresponding filter should be known, in order to actuate the trigger for events, as well as the concrete plain text, in order to load the exact information in the hash function. Otherwise, they can not progress orchestratedly in an ensemble.

The purpose of the task is to find a username and a password that make the function CheckHashVerify return true.

Leakage of entropy

The first thing, which can be seen in the above function, suggesting, that it would be easy to break, is the leakage of entropy. Irreversible operations are used, which reduce the number of the states in which the system could be.

For facility's sake here is given a spread out version of the internal loop, with multiplication by 6, factored down and division by 3, followed by inverse multiplication.

```
a *= 2;
a *= -3;
a += b;
a += 0x74FA;
```

```
a -= e;
b -= b % 3; // rounding to a multiple of 3, towards
0
b *= -1431655765; // reciprocal of 3 (mod 2^32)
b += a;
b += 0x81BE;
b -= e;
```

When working in modular arithmetic, some multiplications are reversible (i.e. there is no leakage of entropy), but others are not.

Multiplying a 32-bit integer by 3 does not increase the amount of entropy because it is reversible. Each input state corresponds to exactly one output state. The operation may be performed also backwards by multiplying with the reciprocal of 3. The reciprocal of 3 is $3^{-1} = -1431655765\ (mod\ 2^{32})$ because the multiplication of both gives a result equal to one: $3 \cdot 3^{-1} = 3 \cdot -1431655765 = -4294967295 = -2^{32} + 1 \equiv 1(mod\ 2^{32})$

Multiplying by 2 is NOT reversible. It does not increase the amount of entropy. This is like that because $(x + 2^{31}) \cdot 2 = x \cdot 2 + 2^{32} \equiv x \cdot 2(mod\ 2^{32})$, in other words both inputs x or $x + 2^{31}$ are collided into the single output of 2 . x. In the worst case thus is limited the possible number of output states to be half the number of the input states, by destroying 1 bit of entropy. Many inputs correspond to one output, so the operation is not reversible and there is a leakage of entropy.

The other irreversible operation is the rounding to the nearest multiple of 3 towards 0. In the worst case this destroys about 1.5 bits of entropy, reducing the number of possible states to about one-third.

These leaks occur to each separate rotation and it is possible their cumulative effect to be very bad. The state is similar to the problems of the type "mixing tank" that are solved, when are studied differential equations, except that the input mixture continues to change the color. If the tank is leaking then the contribution of the early colors to the average color decreases exponentially, rather than linearly, as more colors are added.

These leakages suggests that the earlier values are in danger of "dilution". Each rotation destroys several bits and replaces them with mixtures of the remaining entropy. Later values do not get destroyed and mixed a lot, but the early ones do. Maybe, in order to find an original, should be looked only the last few characters instead of all. For finding a collision could be added the same long suffix to each two starting strings.

It appears that these leaks are not catastrophic, but they really should not have existed in the first place. The fixing of the leak, caused by multiplying by 6, is as easy as changing 6 to 7. The fixing of the leak, caused by rounding to a multiple of 3, is just as easy: the rounding is simply removed. In fact the last idea is *terrible.*

Almost linear

All operations in the hash function, with the exception of rounding to a multiple of three, are linear. They are allocated on addition.

If the operation rounding is removed, the participations of each input can be separated and reduced to a separate multiplicative constant that depends only on the position compared to the end of the string. Each input value is multiplied by the constant, corresponding to that position, the products will be summed and this is the result of the hashing. Finding an input that hashes a given value, would have solved the problem with the sum of the subsets.

An interesting fact: If the leak of entropy due to the rounding is fixed (by eliminating it), but the leak from the multiplication by 6 is not fixed, the things will become even worse. The constants, corresponding to the positions, will obtain coefficients of two. In the end, only the last four characters will have non-zero corresponding constants and the collisions will be slightly easier to find.

It is interesting that the operation, rounding b to be a multiple of three, affects the state to a very small extent. It offsets it at most by 2, but this is the only reason due to which the reversal of the one-way encryption function is difficult. Of course, at suitably designed hash function, the nonlinearities are reversible (e.g. can be applied XOR a into b instead of adding a into b, probably reversing half of the b bits).

The fact that the nonlinearity is so small suggests that there can be applied an integer programming to the problem. Probably the solutions with integer constants are much faster when there is this type of

regularity. In fact the idea does not appear to be a good one.

The solutions with integer constants are not designed for modular arithmetic. Each used solution fails to reverse even three of the seventeen rotations, necessary to process a separate symbol, because are required values that exceed the valid range of the solvers. The confusing in the case is that appears only the message "no solution". The only solver that gives an indication that the error is due to going out of the range, is IBM CPLEX.

A new approach

After several failed attempts it is logical to try also the most obvious things, namely to seek the answer with brute force.

First, lets just list all inputs. This is done very slowly after receiving five characters, since there are 93 possibilities for each symbol and $93^5 = 6956883693 \approx 10^{10}$. At so many possibilities for checking, each additional operation needed to check a separate possibility, adds at least a second to the time of operation (and the hashing includes hundreds of operations). At six characters the time is already one hundred seconds per operation, i.e. it may be necessary to wait for days.

Second, let's try something in the middle.

Since the integrity of the state of the one-way encryption function is used as an output, it is possible to run it reversibly backwards (however, this is slower). You simply have to make inversions of each operation. In this way can be examined both forward and backwards, while finding common middle states.

To say that this gives an increase of the performance is simply an understatement. Instead of using almost a trillion hash operations for all possible six character strings, will be necessary only million hash operations and million reverse hash operations. These million hash operations are used for testing of all possible three character prefixes, building a dictionary that takes a reached state and gives the prefix that has reached it. The reversed hash functions test all possible three character suffixes, by telling which intermediate states can be reached by examining backwards from the final state. If there is a way from the start point to the end point then each state, reached by going backwards, will be in the dictionary and this will bring an end to it.

Third, let's use a bloom filter instead of a dictionary for storing the middle states. Instead of immediately obtaining a solution when finding a match in the middle, each match is a possible solution, which can be checked later by re-examining the possible prefixes.

Why it is worth sacrificing the immediate result, in order to go from three "cached" rotations to four cached rotations? Because each cached rotation is effectively a 100-fold speedup.

Fourth, let's trace the integer restrictions. If they are many restrictions known, which the intermediate states have to satisfy, then they can be checked constantly and to be rejected states that do not fit. When it is measured by how much this reduces the space for searching, it turns out that it is 50% per reverse rotation.

At this stage is detected the first result. One of the user names has only seven characters: "Procyon". And still the problem with the time remains, because the verification of all restrictions lasts for very long.

In fact 50% reduction of the space for searching from the restrictions is wrong. In fact the restrictions simply catch what would have been caught at the next inverse multiplication or reverse division by 3. In other words, the limitations reach 0% reduction. Their elimination speeds up the things quite a bit, by allowing the searching of all 9 character strings.

Finally, let's switch the direction of caching. The going backwards is much more expensive than the going forward, and there is a restriction of the memory for caching of less rotations. The caching of results of going forward instead of going backwards reduces the amount of reverse hash operations and makes possible the searching of all strings up to ten characters, as long as you wait for several days.

Collision

After all, there is a weakness found that is used for beating the hash function: The size of its output.

The size at the output is 64 bits, allowing a little more than 10^{19} possibilities. All strings up to ten

characters (at 93 possibilities per symbol) can be searched through, which makes 93^{10} possibilities. This is around five times 10^{19}.

At this time it is not important how long is the actual password. By pure luck of the brute force can be found strings, which are hashed in the same way.

Code

The following code is used for breaking the cryptographic hash function:

```
/// Returns a given initial state and a sequence of values with the given length that reach the given end state.
/// If such a sequence does not exist, it returns null.
public static HashTuple<HashState, int[]> HashBreak(HashState end,
                                int fictiveAssumedLength,
                                IEnumerable<HashState> initiallyStates) {
   // Generates a Bloom filter backwards from the end
   var numReversiveExpandBackward  = (fictiveAssumedLength -
 1).Min((fictiveAssumedLength * 2) / 3).Max(0).Min(4);
   var filter = HashStateBloomFilter.GenReverseCache(end, numReversiveExpandBackward , pFalsePositive: 0.0001);

   // Examine forward from the initial states to the filter, rejects the states that do not match
   var filteredPossiblePartialSolutions  =
      from start in initiallyStates
      from middleStateAndData  in start.ExploreWholeTraceVolatile(fictiveAssumedLength - numReversiveExpandBackward )
      where filter.ProbablyContain(middleStateAndData .Item1)
      select new { start, data = middleStateAndData .Item2.ToArray(), end = middleStateAndData .Item1 };

   // base case: Insufficient length to meet in the middle. The parts are in fact complete solutions.
   if (numReversiveExpandBackward  == 0) {
      return filteredPossiblePartialSolutions
         .Select(e => HashTuple.Create(e.start, e.data))
         .GetFirstOrDefault();
   }

   // it is not desirable to wait for all possible partial solutions before checking. That would take tons of memory.
   // It is not desirable to be made a verification after each possible partial solution, because that is expensive.
   // so the possible solutions are separated and verified, when there is enough such, to be worth the time.
   var parts = filteredPossiblePartialSolutions .PartVolatile(10000);

   // Completion of all partial solutions
   var solutions =
      from part in parts
      let partialSolutionMap = part.ToDictionary(e => e.end, e => e)
      // Recursive solving of the difference
      let secondHalf = HashBreak(end, numReversiveExpandBackward , partialSolutionMap.Keys, true)
      where secondHalf != null
      // Everything, which is reaching up to here, is a solution. To be combined with the first half and to be returned.
      let partialSolution = partialSolutionMap[secondHalf.Item1]
      let start = partialSolution.start
      let data = partialSolution.data.Concat(secondHalf.Item2).ToArray()
      select HashTuple.Create(start, data);

   // Running the queries
   return solutions.GetFirstOrDefault();
}
```

The above code does a bloom filter containing states that may reach the end by adding a suffix with some length (up to 4). Then it iterates the prefixes of the extra length and notes each one that matches the filter. As soon as it reaches ten thousand corresponding prefixes or ends, it tries to break recursively the difference from the states, reached by comparing prefixes against the end state. If it finds a way to break the difference, the correct prefix forms a pair with the solution of the difference, in order to form a complete solution. Otherwise, it continues, until the prefixes run out.

It should be noted that the code is not optimized much. More precisely, Linq queries are used instead of the equivalent imperative code. Neither the compiler of C#, nor the .Net jit optimize them well and so the code pays for tones of calls of the virtual function, even when it is not necessary. On the other hand, the equivalent imperative code is difficult to use, because in the end, everything is mixed up together in one big mess.

Solutions

After completing the computations the code returns a password, which corresponds to the hash of the password. The password is " < +mt1BmgbNht" (or rather <+ mt1BmgbNht is a string, that hashes to the same thing as the real password). It is seen that the password has 12 characters, but this is due to the fact that the first two characters of the password are omitted in the hash code, i.e. only 10 characters are sought.

3. CONCLUSION

Useful things when writing hash functions:

- There shall be no leakage of entropy. All operations on rounding should be reversible.
- The entire state of the hash must not be used as a result. The going back from the result can be difficult.
- Use non-linear combinations of operations and apply them often. The results at each input must be difficult for separation.

A result with many bits must be obtained. The collisions must be difficult to detec

REFFERENCES

[1] E. Biham, A. Shamir. Differential Cryptanalysis of the Data Encryption Standard, Springer-Verlag, 1993.
[2] E. Biham, R. Chen, Near collision for SHA-0, Advances in Cryptology, Crypto'04, 2004, LNCS 3152, pp. 290-305.
[3] B. den. Boer, A. Bosselaers. Collisions for the compression function of MD5, Advances in Cryptology, Eurocrypt'93 Proceedings, Springer-Verlag, 1994.
[4] F. Chabaud, A. Joux. Differential collisions in SHA-0, Advances in Cryptology, Crypto'98 Proceedings, Springer-Verlag, 1998.
[5] S. Cotini, R.L. Rivest, M.J.B. Robshaw, Y. Lisa Yin. Security of the RC6 TM Block Cipher, http://www.rsasecurity.com/rsalabs/rc6/.
[6] I. B. Damgard. A design principle for hash functions, Advances in Cryptology, Crypto'89 Proceedings, Springer-Verlag, 1990.
[7] Nikolay Raychev. Quantum circuit for spatial optimization. International Journal of Scientific and Engineering Research 06/2015; 6(6):1365-1368. DOI:10.14299/ijser.2015.06.004, 2015.
[8] Nikolay Raychev. Encoding and decoding of additional logic in the phase space of all operators. International Journal of Scientific and Engineering Research 07/2015; 6(7): 1356-1366. DOI:10.14299/ijser.2015.07.003, 2015.
[9] Nikolay Raychev. Measure of entanglement by Singular Value decomposition. International Journal of Scientific and Engineering Research 07/2015; 6(7): 1350-1355. DOI:10.14299/ijser.2015.07.004, 2015.
[10] Nikolay Raychev. Quantum algorithm for spectral diffraction of probability distributions. International Journal of Scientific and Engineering Research 08/2015; 6(7): 1346--1349. DOI:10.14299/ijser.2015.07.005, 2015.
[11] Nikolay Raychev. Reply to "The classical-quantum boundary for correlations: Discord and related measures". Abstract and Applied Analysis 11/2014; 94(4): 1455-1465, 2015.
[12] Nikolay Raychev. Reply to "Flexible flow shop scheduling: optimum, heuristics and artificial intelligence solutions". Expert Systems; 25(12): 98-105, 2015.
[13] Nikolay Raychev. Classical cryptography in quantum context. Proceedings of the IEEE 10/2012, 2015.
[14] FIPS 180-1. Secure hash standard, NIST, US Department of Commerce, Washington D.C., Springer-Verlag, 1996.
[15] RIPE. Integrity Primitives for Secure Information Systems. Final Report of RACE Integrity Primitives Evaluation (RIPE-RACE 1040), LNCS 1007, Springer-Verlag,
1995.
[16] R.C. Merkle. One way hash function and DES, Advances in Cryptology, Crypto'89 Proceedings, Springer-Verlag, 1990.
[17] R.L. Rivest. The MD4 message digest algorithm, Advances in Cryptology, Crypto'90, Springer-Verlag, 1991, 303-311.
[18] R.L. Rivest. The MD5 message-digest algorithm, Request for Comments (RFC 1320), Internet Activities Board, Internet Privacy Task Force, 1992.