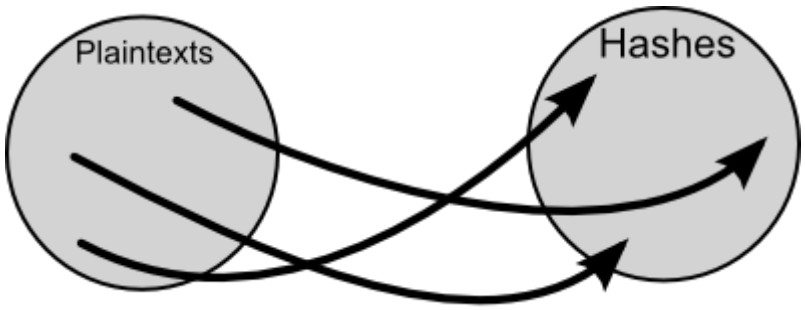
kestas.kuliukas.com

How Rainbow Tables work

I found the creator of Rainbow T able's paper, aimed at cryptanalysts, was pretty inaccessible considering the simplicity and elegance of Rainbow T ables, so th is is an overview of it for a layman.

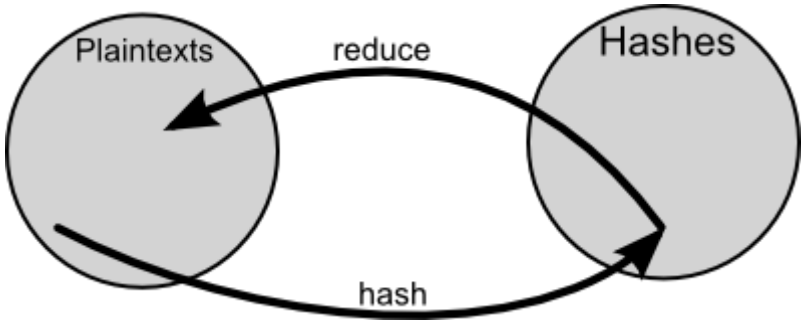
Hash functions map plaintext to hashes so that you can't tell a plaintext from its hash.



If you want to find a given plaintext for a certain hash there are two simple methods:
- Hash each plaintext one by one, until you find the hash.
- Hash each plaintext one by one, but store each generated hash in a sorted table so that you can easily look the hash up later without generating the hashes again

Going one by one takes a very long time, and storing each hash takes an amount of memory which simply doesn't exist (for all but the smallest of plaintext sets). Rainbow tables are a compromise between pre-computation and low memory usage.

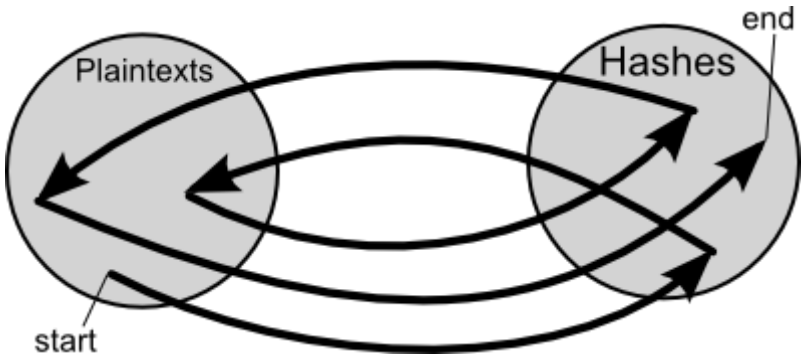
The key to understanding rainbow tables is understanding the (unhelpfully named) reduction function.
A hash function maps plaintexts to hashes, the reduction function maps hashes to plaintexts.



It's important to note that it does the reverse of a hash function (mapping hashes to plaintexts), but it is /not/ an inverse hash function. The whole purpose of hash functions is that inverse hash functions can't be made. If you take the hash of a plaintext, and take the reduction of the hash, it will not give you the original plaintext; but some other plaintext.

If the set of plaintexts is [0123456789]{6} (we want a rainbow table of all numeric passwords of length 6), and the hashing function is MD5(), a hash of a plaintext might be MD5("493823") -> "222f00dc4b7f9131c89cf f641d1a8c50".
In this case the reduction function R() might be as simple as taking the first six numbers from the hash; R("222f00dc4b7f9131c89cf f641d1a8c50") -> "222004".
We now have generated another plaint ext from the hash of the previous plaintext, this is the purpose of the reduction function.

Hashes are one-way functions, and so are reduction functions. The chains which make up rainbow tables are chains of one way hash and reduction functions starting at a certain plaintext, and ending at a certain hash. A chain in a rainbow table starts with an arbitrary plaintext, hashes it, reduces the hash to another plaintext, hashes the new plaintext, and so on. The table only stores the starting plaintext, and the final hash you choose to end with, and so a chain "containing" millions of hashes can be represented with only a single starting plaintext, and a single finishing hash.



After generating many chains the table might look something like:
iaisudhiu -> 4259cc34599c530b1e4a8f225d665802
oxcvioix -> c744b1716cbf8d4dd0f f4ce31a177151
9da8dasf -> 3cd696a8571a843cda453a229d741843
[...]
sodifo8sf -> 7ad7d6fa6bb4fd28ab98b3dd33261e8f

The chains are now ready to be used. W e have a certain hash with an unknown plaintext, and we want to check to see whether it is inside any of the generated chains.

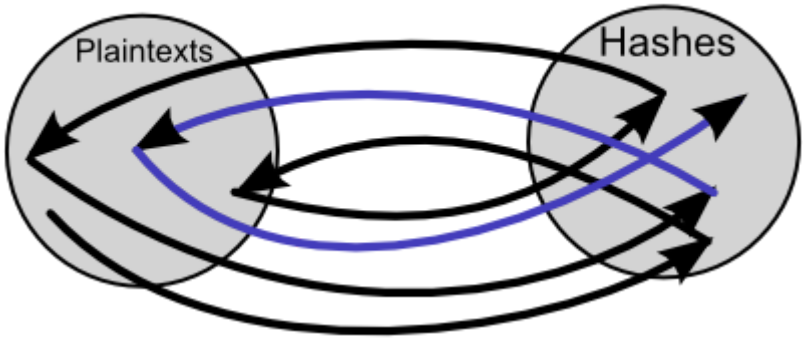
The algorithm is:

- Look for the hash in the list of final hashes, if it is there break out of the loop.
- If it isn't there reduce the hash into another plaintext, and hash the new plaintext.
- Goto the start.
- If the hash matches one of the final hashes, the chain for which the hash matches the final hash contains the original hash.

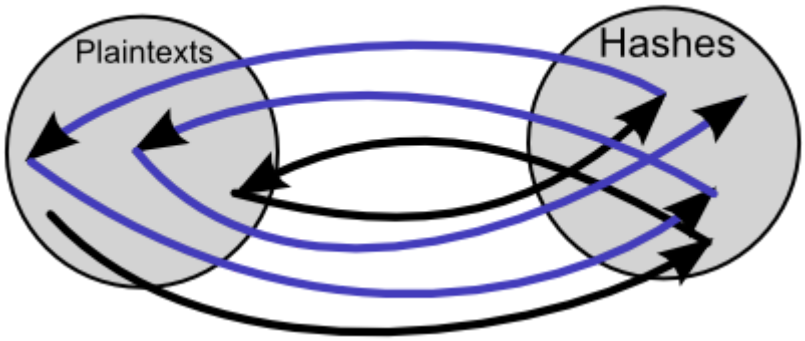
You can now get that chain's starting plaintext, and start hashing and reducing it, until you come to the known hash along with its secret plaintext.

In this way you check through the hashes in the chains, which aren't actually stored anywhere on disk, by iterating column by column through the table of chains, backwards from the last column in the chain, to the starting plaintext.

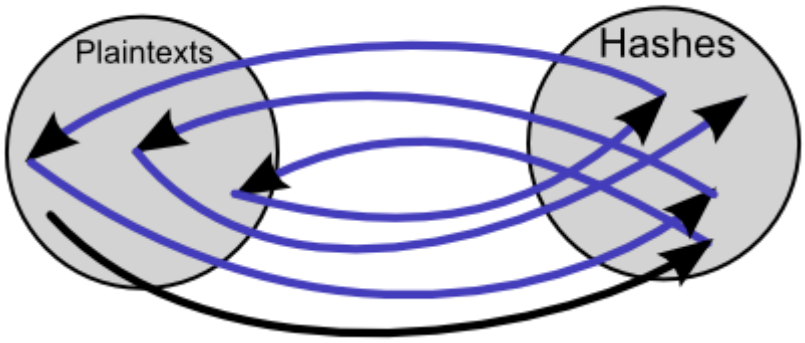
If you wanted to check whether the hash exists in the last column of any of the chains you reduce and hash the given hash once, then check the generated hash against the chain end hashes.



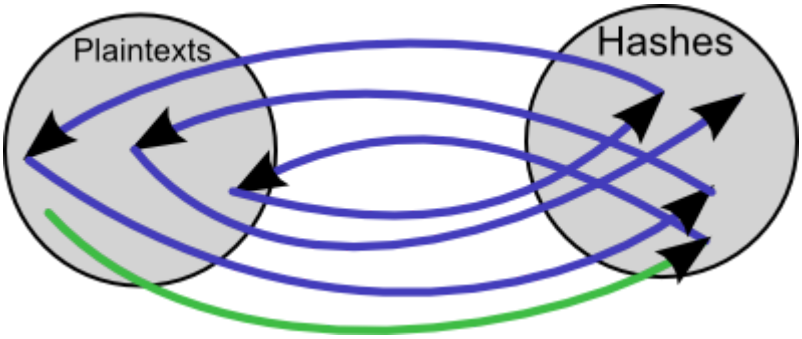
You can check the second last column by reducing and hashing twice, then check the generated hash against the chain end hashes.



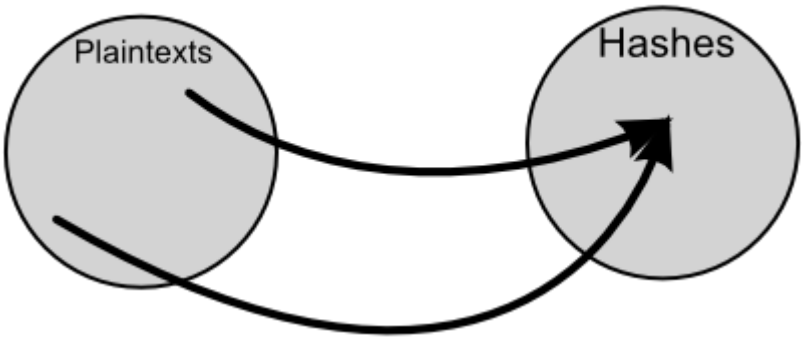
And the third is checked by reducing and hashing three times, then check the generated hash against the chain end hashes.



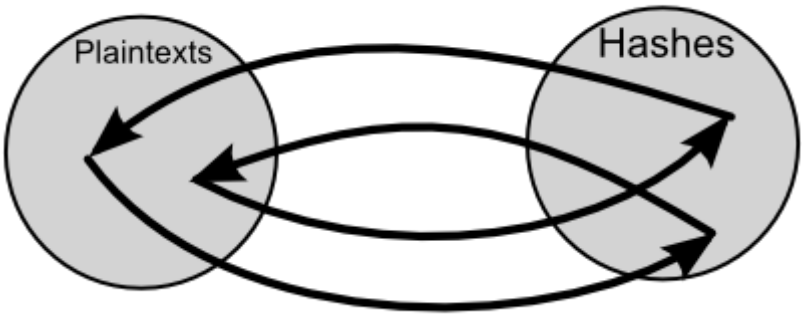
Supposing a chain ending matches the generated hash the matching chain end might contain the hash. The starting plaintext which was stored with the ending hash can be reduced and hashed until the correct plaintext is found within the chain.



Collisions are the only problem with Rainbow T ables. Ironically collisions are s een as a bad thing for hashing algorithms, but in the case of Rainbow T ables a hashing algorithm which generates collisions fairly regularly will be more secure.



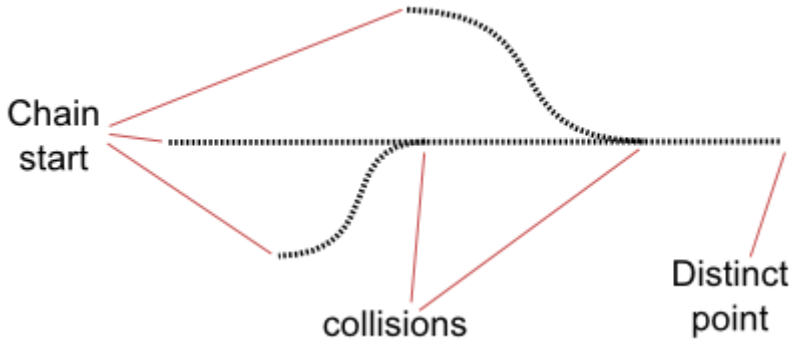
A given hash may be generated by multiple plaintexts (this is called a collision), which is a big problem for chains because it causes chains which start dif ferent to converge into one. Also you get loops, which are caused when a hash is reduced to a plaintext that was hashed at a previous point in the chain.



Because of these collision problems there is no guarantee that there will be a hash of a plaintext that will reduce to some other given plaintext.
If you have a simple list of hashes and corresponding plaintexts for every plaintext in a set you will know that if you have not found the hash in the generated hashes the plaintext that generated the hash is not in the set.
If you have a table of chains where the reduction function reduces hashes into the set of plaintexts you could have trillions of chains generated but you still may not have generated every plaintext in the set you want to check. Y ou can only say how probable it is that a table of chains contains a certain plaintext, and this can approach 1 but will probably never reach 1.
If you have a rainbow table with 10 chains of length 100 you have hashed 1000 plaintexts, but even if there are only 100 plaintexts in the set of desired plaintexts the 1000 hashes you have in the chains may not contain all the desired hashes.

The way collisions are handled is what sets Rainbow T ables apart from its pre decessor which was developed in 1980.

The predecessor solved the problem of certain plaintexts never being reduced to by using many small tables. Each small table uses a dif ferent reduction function. This doesn't solve the problem completely , but it does help.
To solve chain merges and loops each chain ended at a "distinct point"; a has h which was unique in some way , eg hashes where the first 4 characters are 0. The chains keep on going until it reaches a distinct point. If two chains end up at the same distinct point then there has been a collision somewhere in the chain, and one of the chains is discarded. If a chain is generated for an unusually long time without reaching a distinct point a loop is suspected (where a chain of hashes ends up reducing and hashing to a previous hash in the chain). The problem with this is tha t if there is a collision there is potentially a whole branch which has to be cut of f and won't make it into the chains, and a loop will cause all the hashes which came before the loop in the chain to be discarded.



Also all the time spend generating that chain will be wasted, and by ending only at distinct points you have chains of variable length. This means that you may have to keep checking for a hash within especially long chains long after the other chains have ended.

Rainbow tables dif fer in that they don't use multiple tables with dif ferent reduction functions, they only use one table. However in Rainbow T ables a dif ferent reduction function is used for each column. This way different tables with dif ferent reduction functions aren't needed, because dif ferent reduction functions are used within the same table. It is still unlikely that all plaintexts in the desired set will be hashed, but the chances are higher for a given number of chains. Chain merges are much, much rarer , because collisions have to occur on the same column. For a chain of length l the chance of a collision causing a merge is reduced to 1/l. Loops are also solved, because if a hash in a chain is the same as a previous hash it won't reduce to the same plaintext.

The reason they're called Rainbow T ables is because each column uses a dif ferent reduction function. If each reduction function was a dif ferent color , and you have starting plaintexts at the top and final hashes at the bottom, it would look like a rainbow (a very vertically long and thin one).
By using Rainbow T ables the only problem that remains is that you can never be certain that the chains contain all the desired hashes, to get higher success rates from a given Rainbow Table you have to generate more and more chains, and get diminish ing returns.

I hope by explaining the Rainbow T able I haven't made them any less wonderful ...

An easy way to improve on the "rainbowcrack" Rainbow T ables impleme ntation

This section probably goes a bit beyond where a layman would be comfortable, but if you're interested in the practical applications of the above theory or have some interest in cryptography read on..

The rainbowcrack application is how most people come to learn about Rainbow Tables, because it is the application which puts the theory above into code. It has been very successful, with many websites dedicated to generating rainbowcrack hash tables and letting users search them.

However there is a pretty clear way this application could be improved, very easily, in the sense that the generated tables would take up a lot less disk space, but be equally as effective for breaking hashes:

Remember above that when you want to generate a certain chain you start from an arbitrary hash. This just means it doesn't matter where you choose to start from. The rainbowcrack application starts from a randomly generated 64-bit number. This number is then used to generate a chain which ultimately ends with a 128-bit hash, which is reduced to another 64-bit number.

Why use a randomly generated number as the starting point? A pseudo-random number generator can generate a fantastic amount of seemingly random numbers from a single input number. Why not make a single random input number, and then store the index of the number which generates the pseudo-random number?

So for example a cipher like RC4 is a pseudo-random number generator. Say the single input number (the "seed", as it's called) was 18092398. The first 64-bits the RC4 generates might give a number of "091358029384092", to start the chain of f. The second 64-bits might give a number of "123793582983480", to start the second chain of f. The third 64-bits might give "1089324083486", for the third chain, and so on potentially for billions of chains.

What is the difference between this and storing a random 64-bit number for each chain, as rainbowcrack does?
Simply that a start-point in a rainbowcrack table must be stored as the randomly generated 64-bit number. A start-point using a random-number generator needs only the single input number (the "seed") and the chain number. So when referring to the third chain in the example above, if you wanted to know the start point of "1089324083486", you would only need to know the "seed" number, and that it was the third 64-bit number generated. That's the number "18092398", and the number "3".
To know the start-point for the fourth chain you only need to know the "seed" ("18092398"), and the number "4".

If you have 2^64 chains (1,844,674,407,370,955,616) then it wouldn't make any difference, but that would be 4194304 terrabytes, far larger than any Rainbow Table ever generated. For a more realistic rainbow table with, say, 2^28 (268,435,456) chains you would only need a 28-bit number instead of storing a 64-bit number, as rainbowcrack currently does.
That's an improvement from (64-bit+64-bit) per chain to (28-bit+64-bit) per chain, plus a single 64-bit "seed" number per table. When you're talking about millions of chains that's a very significant reduction of data for the same hash-breaking ability.

In this example a rainbowcrack table would be 2^28 * (64-bit random start number + 64-bit chain-end number) (4096 MB).
Using a pseudo-random number generator the table would be 2^28 * (28-bit non-random start number + 64-bit chain-end number) + 64-bit "seed" number (3264 MB)

When you scale that difference up to the huge sizes rainbowcrack tables can reach the savings become massive, and you end up with whole hard-disk arrays of randomly-generated chain-start number data that is pure waste, not to mention the bandwidth used moving the data around.
A huge waste considering the trivial code changes.