Brad Israel
bdi8241@cs.rit.edu
Research and Individual Project
02/22/08

Rainbow Tables/Time-Memory Trade-off Techniques

Abstract

The goal for this project was to research time-memory trade-off techniques, as they relate to cracking Windows LanMan hashes, and then implement a framework that is more useful for academic purposes. The framework is written and documented in Java so that the code is easier to understand and follow and so that it is easy to modify or replace elements of the algorithm, such as the reduction function. The framework will also allow the user to generate statistics and benchmarks before or/and after generating and searching the tables, which should allow the user to compare their modifications in terms of accuracy and performance for Windows password cracking. This should make rainbow tables easier to understand and "play" with for a wider audience.

All code and other useful links can be found here:
http://www.cs.rit.edu/~bdi8241/TMCrack/

History

The concept of a time-memory trade-off has been around since 1980, when Martin Hellman published a paper (see works cited) detailing a general technique for storing and searching large datasets in memory. He developed the technique for use with solving discrete logarithms or other such number theoretical problems. Then in 1982, Ron Rivest modified the technique to use distinguished points as the end of the chain. This modification put a better upper bound on the number of calculations needed when searching the tables to t^2, where t is the length of the chain. Twenty years later, in 2003, Phillipe Oechslin developed a new chain structure, which he calls rainbow chains, that allows for static chain lengths and an improved calculation upper bound of about $t^2/2$. His paper also generated a lot of interest on the topic because of his focus on using it against a practical target, Windows passwords, and his website even included a live demo where people could submit hashes to see if they could be cracked using his applied technique.

Theory

The goal of using rainbow tables instead of a brute force tool, like John the Ripper, is to get close to the same results, ~100% of hashes cracked, but not have to compute the brute force values every time you want to break a new set of passwords. The problem is that you can't efficiently store all of the hashes (8 bytes) with their respective passwords (7 bytes, assuming maximum password length) because it would take about (36^7)*8*7 bytes, or 4087Gb of hard disk space for alpha-numeric passwords. Rainbow tables are the best of both worlds, you have to do much less computation, but because you are doing some of the computation when trying to crack a password, you now have to store less data on the hard disk. This is why it is considered a time-memory trade-off technique, the more you store on the drive, the less time it takes, but requires more memory and vice-versa.

A rainbow table just contains a long list of start and end points for different rainbow chains. A rainbow chain is what makes this technique practical because if you generate a chain of length 1000, you can theoretically store 1000 hash to plaintext pairs by storing the start and end of the chain. Since the start and end of the chain are stored as two 8 byte entries, it gives you the ability to break 1000 hashes using 16 bytes of data. The main algorithm behind the rainbow chains is called a reduction function. A reduction function basically attempts to map a hash value back to a unique plaintext value.

This function obviously doesn't work perfectly (one-to-one mapping with hashes to plaintexts) because doing so would imply that the hash function used was reversible, which means that the hash function has been broken and there is no need to deal with rainbow tables because you can just reverse any hash. This imperfection is what causes many false alarms when searching for a hash because even though you found the end point, when you search through the chain, the hash still might not show up, but can be found in a later chain with the same end point. The chains are generated, either for generating or searching the tables, with the following equation:

$$R_t(LM(R(LM(R(LM(R_0(LM(P_0)))))))) = P_t$$

Where R is the reduction function, LM is the actual Windows LanMan hash function, the '0' subscript is the beginning of the chain, the 't' subscript is the chain length or end of the chain, P_0 is the initial plaintext (either randomly generated or found using the reduction function), and P_t is the final plaintext. After generating the chain, you would store P_0 and P_t in memory or to the hard disk. To search the tables you would first reduce your unknown hash and look through the table for a match with any P_t's, then regenerate the chain starting with P_0 to P_t using the same equation, but before moving to the next inner iteration just check to see if the unknown hash matches any of the newly generate hashes. If they do, then you've successfully cracked the password.

The main problems with rainbow tables deal with their imperfect reduction function. False alarms were mentioned before, but there are also a couple others, including collisions and loops. Collisions, also called merging, is when two hashes from different chains reduce to the same plaintext. This causes the chains to converge which causes both the false alarms and the reason that rainbow tables always have a probability of breaking a password (approaches 1, but will never be exactly 1 for any size keyspace). The probability is because, since collisions are inevitable, the chains could miss entire blocks of the keyspace that are not contained in any other chains. The other minor problem is loops in the chains. When a loop occurs, a subset of the chain repeats one or more times when generating the chain, which mostly just wastes CPU cycles and is not a major concern.

Implementation – Reduction function

The reduction function that I used to generate and search my tables was very simple and followed the plain-text example.(Plain-text Basics) It basically xor's the hash with the unique column number (also known as the current position in the chain length, which goes from 0 to t), then uses a modulus and divide method to extract a plaintext from the hash. The algorithm is:

```
ReductionFunction()
        xor the hash with the column number to get X
         loop from max_pass_length to 0:
         extract a character, X mod charset_length
         move to next character, X / charset_length
```

A quick numerical example of the function is:

LM(bcdd) -> 480B3416EB74922E
480B3416EB74922E XOR 1 = 5191300268518838831

5191300268518838831 % 5 = 1
5191300268518838831 / 5 = 1038260053703767766
1038260053703767766 % 5 = 1
1038260053703767766 / 5 = 207652010740753553
207652010740753553 % 5 = 3
207652010740753553 / 5 = 41530402148150710
41530402148150710 % 5 = 0

Returns next plain text = 1130 or bbda (alpha-numeric)

This reduction function is mainly for example purposes and needs to be reworked if you want to actually create a practical version. A good reduction function would be one that reduces the number of collisions, and therefore false alarms, loops, and is still fast to compute.

Implementation – Generating a Table

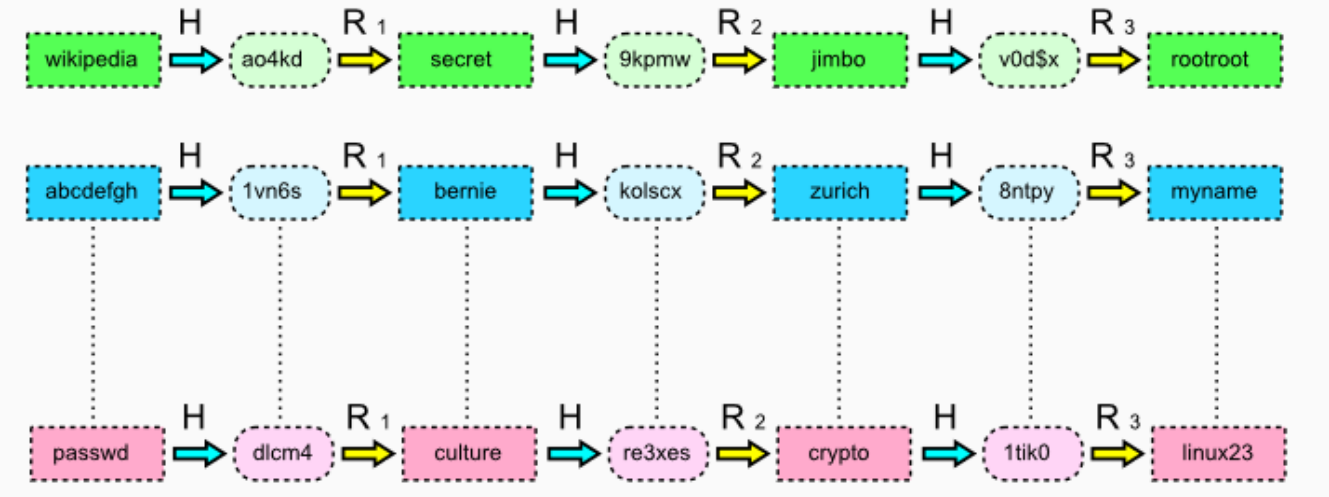
Generating a rainbow table is very similar to a brute force method because it starts at a random plaintext in the keyspace, then applies the equation shown in the theory section. The algorithm will keep looping through this pattern until it outputs the number of chains that you think will be statistically successful for breaking passwords in the entered keyspace. The algorithm I used is:

GenerateTable()

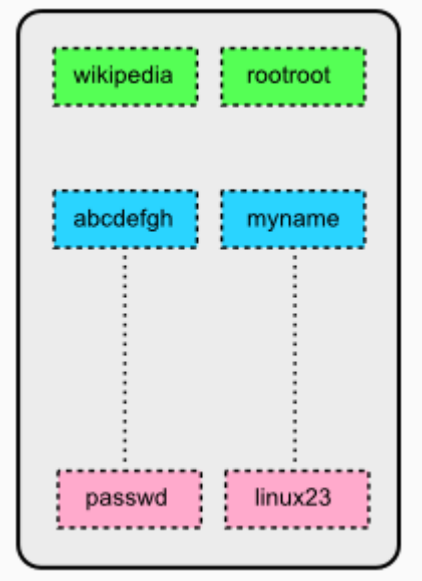
Given a chain length, t
Choose a random plaintext, P, from the keyspace
Hash P using the LM hash function, LM(P) = H
Apply the reduction function, R(), to the hash, H, to get a new plaintext, P_1
Repeat this process from 0 to t(chain length)
Store only P and the last plaintext in the table

For my implementation, the plaintexts are really indices, represented as hex bytes, into the keyspace instead of actual alpha-numeric characters. As an example "0000000000000000 = A" and "000000000005D25A = URJF". This makes it easier to store and look up later.

Wikipedia has a good example of how table generation works:



From the picture you can see the three simplified (chain length of 4) chains and how they are generated. When they are stored in the table it would look like the following, but be able to generate all of the above passwords.

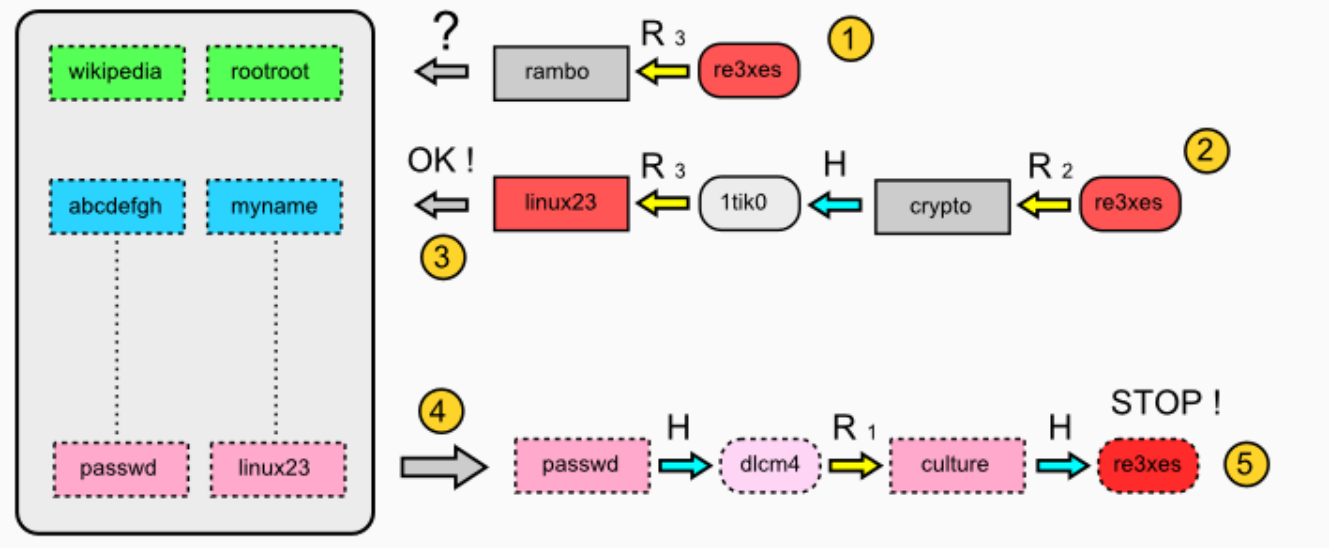


Implementation – Searching a Table

Searching the table is implemented almost exactly opposite to the generation of the table. To search the table, you are first reducing, starting at the last column and working back toward the 0^{th} column, the unknown hash to a plaintext and then searching the ends of the chains for a match. Once a match is found, just regenerate the chain and try to match your unknown hash to a known plaintext's LanMan hash. The algorithm is:

SearchTable()
Given a hash, H
Use last column reduction function to get plaintext, P
Search the table for a matching end of the chain
If nothing matches
Repeat search using last column-1 reduction
Keep repeating with last column-n reduction until it equals the 0^{th} reduction
This means that the password wasn't represented in the table
If found
Regenerate the chain from the start to the end
Before each reduction check if hashes match
If they match, password found
Else loop and try to find another end

Again, Wikipedia has a nice example of this:



The initial unknown hash is “re3xes”, which reduces to “rambo”, but isn't an end point for any chains. The hash is then reduced starting with the 2nd to last reduction function and produces “linux23”, which is an end point. Once it has a start and end point, it regenerates the chain and waits for a plaintext that hashes into the unknown hash, which ends up being “culture”.

Results

My first experiment was a test to see how much faster a real version of rainbow tables, rcrack, were against de facto standard, John the Ripper. I used a default installation of John the Ripper (Linux) and used its default dictionary and mangling rules. For rcrack, I used 64x1Gb (64Gb total) tables that covered the “alpha-numeric-symbol14-space” charset, which covers 99% of allowed windows passwords. I wouldn't recommend trying to generate these tables on your own, as it will probably take a couple years, but they are available for free download online(Hint: Google shmoo rainbow tables).

Password	John the Ripper Time (Mins)	RainbowCrack Time (Mins)
win	0.07	1.25
confidential	0.20	2.58
tmjo	0.07	1.44
kmvceqkguawtbi	2.9	7.71
t3h	0.07	1.28
tq^bj0t#r3	1191.93 (20 hours)	3.03

It's pretty clear that if the password was in John's dictionary or covered by its mangling rules, then it was recovered almost immediately by John the Ripper. Those same passwords took a little longer using rcrack because it had to search through a portion of 64Gb and then still do some generation. Where rcrack really shows how powerful it is, is when you use passwords that are not alpha-numeric and not close to a dictionary word, like the last one. What took John almost a day to break was broken using rcrack in about 3 minutes. This is because it basically reduces the complexity of password cracking to that of a search. Unfortunately, my implementation is very much slower than rcrack's implementation. It took me 5 days to actually generate a useful rainbow table and when cracking passwords, the ones it

is able to break take 1-4 hours. Most of this has to do with Java's speed issues and overhead, which I talk about in the next section. I will definitely work to improve this, so if you're interested, check back on the website for updated versions.

Recommendations for practical implementation

If anyone wanted to implement rainbow tables for a practical project, I have a few recommendations. First, don't use Java. This is for two reasons, Java can't really handle 64 bits (16 bytes) in its long primitive like c or other languages can, using an unsigned long, so when you try and store a full 64 bit hash as a long value it overflows. This caused me to write an UnsignedLong class in my code that split the hash into two 8 byte segments and stores them as “upper” and “lower” byte segments. The other reason not to use Java is because, for these operations, it is very slow. The time it takes to generate chains is much longer than the rcrack, c, implementation. The other recommendation I have is to put a lot more effort into generating a proper reduction function so that it's faster and minimizes the problems that I discussed in the reduction function section.

Future work

What I'd like to do in the future is expand the framework to make it more useful. I'd like it to be able to handle generating and searching through multiple table files because the Oechslin paper points out that this actually increases the keyspace coverage. I'd also like to add more or improve the existing statistics and benchmarks, so that they gave more useful results. And just in general, I'd like to improve the speed of the code and the reduction function, so that it is a little more practical rather than mostly an educational and informative tool.

Effective Policy changes

After researching this technique and seeing it in action, I would definitely change some of the policies concerning password security, if I were in charge of the security policy at some company. I would first make sure that all important applications, that required password identification, used “salted” passwords. This means that the each password's hash would look different even if some of them were the same. This would almost completely defend against this technique because rainbow tables can only be generated for one salt at a time, so if someone wanted to crack a salted password, they would first need to know the salt and then regenerate the table using that salt. At that point it would be easier just run a brute force attack against the password. On Windows machines, I would set a policy that all systems, especially servers, have to turn off LM and NTLM v.1 hashing, which can be done by setting a registry value. This would force the systems to use NTLM v.2, which uses different salts for each password the way Unix/Linux/OSX systems have done it since password shadowing was invented. A neat trick to stop your passwords from being LM hashed on Windows systems that you may not have complete control over is to add non-printing ALT characters to your password. Doing this causes the resulting hash to be incompatible with LM hashing, so it never gets stored insecurely. Also, adding non-printing ALT characters is good in general because most password crackers, including John and rainbow tables, don't even consider non-printing ALT characters as part of the keyspace.

Another way to combat this, if you're working for a company like the NSA or one with high-value trade secrets, would be to add biometrics, secure cards, RSA keys, etc. into the security policy for any systems that absolutely need to be secure.

Works Cited

Papers:
Philippe Oechslin: Making a Faster Cryptanalytic Time-Memory Trade-Off. CRYPTO 2003: 617-630.http://lasecwww.epfl.ch/pub/lasec/doc/Oech03.pdf

M. E. Hellman. A cryptanalytic time-memory trade off. IEEE Transactions on Information Theory, IT-26:401–406, 1980.

Websites:
How Rainbow Tables Work - http://kestas.kuliukas.com/RainbowTables/
Ophcrack - http://lasecwww.epfl.ch/~oechslin/projects/ophcrack/
Parameter Optimization in RainbowCrack - http://www.antsight.com/zsl/rainbowcrack/optimization/optimization.htm
Plain-text Basics - http://www.plain-text.info/Rainbowtables_Basics/
RainbowCrack - http://www.antsight.com/zsl/rainbowcrack/
Rainbow Tables Explained - https://www.isc2.org/cgi-bin/content.cgi?page=738
Wikipedia – Rainbow Table. http://en.wikipedia.org/wiki/Rainbow_table