Jake Haapoja

Computer Science

Saint Cloud State University

Saint Cloud, MN, United States

JLHaapoja@stcloudstate.edu

**Abstract– The objective of this paper is to present an analysis of basic Least Mean Squares (LMS) adaptive filtering as applied to speech recordings with various levels of Additive Gaussian White Noise (AGWN), and how this impacts the transcription of the Audio using pre-trained models from the Natural Language Toolkit (NLTK) python library.**

***Keywords–Speech Recognition, Adaptive Filtering, Least-Mean-Squares, LMS, Natural Language Toolkit, NLTK, Noise Cancellation***

**I. Introduction**

The MD5 message-digest algorithm is a popular hash function that produces a 128-bit hash value. MD5 was designed in 1991 by Ronald Rivest to replace the much less secure MD4 algorithm. The algorithm contains a series of steps including a padding step, a step that breaks the message into 512-bit chunks, a step that breaks these chunks into sixteen 32 bit fragments, and a main loop step.

A common attack used against the MD5 algorithm is known as a rainbow attack. In this attack, a precomputed list of hashes (with their accompanying plaintext passwords) is created using the MD5 algorithm. A rainbow attack takes this list and iterates through it in an attempt to find a match between a given hash and the precomputed rainbow table.

-overview of UNIX password Implementation

We were particulary interested in seeing how the MD5 algorithm would be able to be upheld when using a rainbow table attack, as well as how effective a rainbow table attack would take in cracking the hash value.

**II. Implementation**

Our program implementation has three key pieces. The first piece is the storage of user information. The second piece is the MD5 algorithm and the final piece is creating and cracking passwords via a rainbow table.

**A. User Information**

When a user is added and a password is specified, a unique salt is generated consisting of 6 characters or 6 bytes. This is done by using a random random number generator six times to get an ascii character between 0-255.

Once the salt is created it is appended onto the user input plain text password and it is passed into the MD5 algorithm to get the hash value. The username, salt and hash are all stored internally for each user.

When the program finishes execution any user information that is stored in memory is written to the “passwords.csv” file. The schema of the file is username, salt, and then hash, with new lines separating records and commas separating fields.

When the program starts the “passwords.csv” file is read and any entry in the file is stored in memory. This way entries don’t need to be re-input on successive executions of the program and it allows for testing with the same data.

The second usage of the program is verifying a password. The user will be prompted for a username and a password that exists within the “passwords.csv” file and is read into memory. The input password will be hashed and compared to the hash value that is associated with the input username.

**B. MD5 Algorithm**

The MD5 algorithm takes a message to be hashed and returns the hash value. At the start of the algorithm the message is converted into a string of bits. Each character in the message is converted into its 8 bit representation.

From there padding must be added to the message to ensure that the length of the bitstring is a multiple of 512 and the last 64 bits must be the length of the bit string. The first number padded is always a 1 to indicate that the padding has started and then 0 padding is added until,

(1)

This will ensure that there are exactly 64 bits left for the size of the bit string. It also ensures that the padding will work on any size input message.

Next the length of the original message is converted into its binary representation. The binary length is in little endian format and once it is constructed it is appended onto the end of the bit string.

With the bit string constructed it is next split into 512 bit chunks. From there each 512 bit chunk is split into 16 32 bit chunks which we call fragments. Since each fragment is 32 bits and in binary, we converted each fragment into an unsigned integer for further processing.

The following algorithm is computed for each 512 bit chunk. The algorithm uses predetermined hex values which are used in combination of our fragments to get the hash. The predetermined hex values are,

(2)

The next part of the algorithm is where the permutations of the bit string occur. There are 64 different permutations that occur. For the first 16 permutations the following bitwise operations are used.Where a is initialized to A, b is initialized to B, c is initialized to C and d is initialized to D.

(3)

After this permutation occurs and at the end of each subsequent run the variables are redefined as,

(4)

Where x is a list of 64 integers and y is a list of 64 hex integers. RunCount is the current run from 0-63 which is used as an index into x and y.

*x = { 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21 }*

y[64] = { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee, 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501, 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be, 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821, 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa, 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8, 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed, 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a, 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c, 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70, 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05, 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665, 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039, 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1,0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1, 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 } (5)

The values in the x list are associated with how many times the value in f should be left shifted. The values in y were calculated using the following formula where i is an iterator from 0 - 63.

(6)

It is important to use the same values for x and y instead of random numbers so that it is possible to generate the same hash values from the same plaintexts between subsequent hashes.

For runs 16 to 31 the formula used changes to,

(7)

After each run from 16 - 31 variables are set again as in equation 4.

For runs 32 - 47 the formula is changed to,

(8)

After each run from 32 - 47 variables are set again as in equation 4.

The final runs from 48-63 use the following formula for manipulation,

(9)

The values are saved on final time as in equation 4. These 64 operations are computed 1000 times where the hash output from one of the 1000 runs is used in the subsequent run as the input. On the 1000th execution the hash code is converted from integers into hexadecimal and the value is returned as the final hash.

.

**C. Rainbow Tables**

Rainbow tables are a variant of pre-computed hash tables that trade computation time for space efficiency. By calculating the hash values for a set of known common passwords or text strings in advance, one can compare a given hash value against the hashes in the table and look up the corresponding plaintext.

This is the essence of a precomputed hash table. Such tables rely on hash chains for generation, that is, they will use a password string as a ‘seed’ value, then taking the resulting hash value and applying a reduction function a new plaintext is obtained, which is subsequently fed back into the hashing algorithm.

H(p1) → c1 → R(c1) → p2 → … → H(pn) → cn  (x)

For a chain of length n, some hash function H() and reduction function R()

Repeating this process results in a so-called hash chain. However, using the same reduction function at each stage can result in several problems, the most significant being what are called merges in the chains. This is where the chains generate the same hash or reduction value and then produce duplicates of all subsequent values in those chains. Effectively this wastes computation time, and slows down lookups as they are re-comparing the hash to the same values multiple times.

H(pij+1) → ci,j+1 → R(ci,j+1) (x)

↘

→ pi+1,j+1 = pk+1,j → …

↗

H(pkj) → ckj → R(ckj)

Such mergers and the collisions that cause them can be mitigated by applying a different reduction function to each stage in the chain. This can be as simple as adding an index value that modifies the reduction function just enough to be mathematically distinct. Then for each chain in the table, you apply the reduction functions in a different order.

For example, consider the case of a table of n chains and t links in each chain, and functions H() and Ri() being the hash and reduction functions respectively. The first chain in the table would be formed by applying the hash function to the first seed string, then the first variant of the reduction function, then the hash on the result, followed by the second R() variant, and so on until all t variants have been applied. Then the second chain would be produced similarly, except starting with the second reduction function and wrapping around such that the first reduction function is applied to the final link in the chain. This is repeated for each chain in the table.

H(p1) → c1 → R1(c1) → … → Rn(cn-1) → pn → …

H(p1) → c1 → R2(c1) → … → R1(cn-1) → pn → …

H(p1) → c1 → R3(c1) → … → R2(cn-1) → pn → …

Multiple reduction functions are necessary for the hash table to be considered a rainbow table. Doing so means that even if a pair of chains collide, they would have to collide at the same stage in their generation in order to merge, which is highly unlikely. This means each chain will have unique hashes at each stage and computation of the table, as well as lookups performed on that table are maximally efficient with regards to space and processing power. The resulting rainbow table can then be easily stored in a file for later use.

The attack algorithm is then fairly simple. All that needs to be done is to scan through the table until an identical hash is found, then return the corresponding plaintext. This drastically reduces the time required to crack any and all passwords for whichever hash function the table was generated with.

**III. Results & Analysis**

**IV. Conclusion**

**V. References**

**V. Appendix**

**A. Program Code**

**B. Design**

Driver.cpp

- bool createNewUser(Password& pData)

- bool verifyPass(Password& pData)

- bool rainbowAttack(Password& pData, RainbowTable& rt)

MD5.cpp

- void MD5::padding(string& msg)

- string MD5::toBitString(string msg)

- string MD5::decToHexa(unsigned n)

- unsigned MD5::lcs(unsigned base, unsigned shift)

- string MD5::hash(string msg)

- string MD5::UIntToBitString(unsigned x)

- string MD5::binToHex(string binary)

- string MD5::crypt(string msg)

Password.cpp

- Password::Password()

- bool Password::readPasswordFile(string file)

- bool Password::writePasswordFile(string file)

- bool Password::verifyPassword(string uid, string pWord)

- string Password::makeSalt()

- bool Password::addUser(string uid, string pWord)

RainbowTable.cpp

- RainbowTable::RainbowTable(int chainNum, int chainLength)

- void RainbowTable::createTable(Password& passData)

- string RainbowTable::reduce(string hash, int index)

- bool RainbowTable::attack(string hash)

- void RainbowTable::readInputSpace(vector<string> &inputs, string fileName)

- void RainbowTable::writeTable(string filename)

**C. Testing**

**D. User Manual**

There are 7 different files required for execution of this program, RainbowTable.h, RainbowTable.cpp, Password.h, Password.cpp, MD5.h, MD5.cpp and Driver.cpp. There are three not required files but they have been provided for easy use of the program. The first is the password.csv file, this contains pre-input user information. The second is the rainbowTable.txt. This file contains a rainbow table with the information stored in the given password.csv file. The final file is our list of potential passwords called, PotentialPasswords.txt. All of these files are packaged into our project zip file.

To compile the code, move the zip file onto a linux terminal and unzip it. Use the following command to compile all the files in the current directory into a binary,

g++ -std=c++11 \*.cpp -o rainbow

To execute the program then use this command,

./rainbow

When the program is executing, there are six options displayed: add a user, verify a password, create a rainbow table, load a rainbow table, perform rainbow attack and finally exit.

Adding a user prompts you for a username and password which generates salt and is hashed. These values are then stored in the password.csv file.

The verify password option requires a username and a password to verify a hash. For simplicity you can use the user name “nathan” and the password “taylor” as they are already in the password.csv file.

A precomputed rainbow table and userdata file has been provided so it isn’t necessary to make a new one as it takes a long time to create. so instead select the load a rainbow table option.

Performing a rainbow attack requires inputting the hash value of an existing hash in the password.csv file.

Finally exit will close the program and save the user data to the passwords.csv file.