Neelam Agrawal
Rodney Beede
Yogesh Virkar
CSCI5576 – High Perf Sci Computing, University of Colorado
Final Report - 2011-05-04

Parallel Zip Archive Password Recovery (P-ZAPR)

Introduction

The goal of our project was to test the performance of a software program designed to run on multiple CPUs in a parallel fashion with the purpose of recovering an unknown password to a zip archive file that has been encrypted with the AES-256 encryption cipher. Strong encryption ciphers such as AES-256 have become part of this commonly used data format for securing private data. This stronger protection is very useful when one does not wish for their data to be accessed by unauthorized parties, but it also creates a much more challenging problem when someone forgets the password effectively losing their data. Our project addresses this problem by using the currently best known method of decrypting the data which is guessing the password. Since the number of password permutations is large we decided to take two approaches: brute force of all 7 character or less alphanumeric passwords and guessing the password from a large number of words in a dictionary.

Our primary focus was to test both methods to calculate the type of performance we could obtain when using varying numbers of nodes on two different supercomputer clusters. By measuring the number of passwords per second that could be processed per node and the total number of passwords we could iterate through in a given amount of time we developed a formula that can estimate the amount of time it would take to discover a password with specific characteristics for an encrypted zip file. This provided useful information on the amount of time

required to find a password of a given complexity as well as underscoring the importance of strong password choices to ensure data security.

Framework

Our parallel implementation of the code was written using C++ and MPI C APIs. The primary division of code components was in three parts: the decrypt engine, the brute force password generator, and the dictionary password generator. When run the program picks one of the generators to use based on a command line argument given by the user. In addition the encrypted zip file is provided on the command line as well as the file containing the dictionary. At start up the main method initializes the decrypt engine by providing the zip file and initializes the password generator with appropriate arguments such as the process's rank, the total number of processes, and the dictionary file if applicable. This allows the generator to evenly divide the range of passwords it will be attempting. The generator also handles the case where the passwords don't divide evenly for the number of processes by distributing the remaining among the first set of processes.

After initialization the main method begins a loop that proceeds as shown in the following diagram:

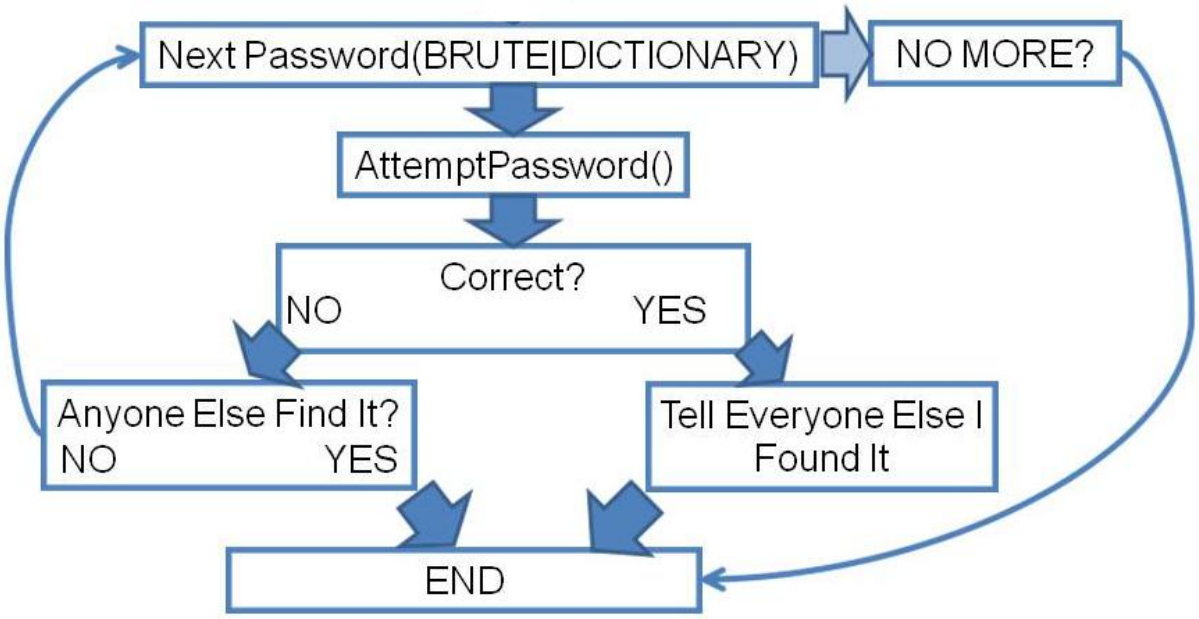


Figure 1. Flowchart for primary worker loop in main method

The process continues attempts until:

- It finds the solution
- Another process notifies it that it found the solution
- It runs out of passwords to try for its range

In addition to attempting the password a log file is maintained which contains a date and time stamp along with the number of attempts made so far. To avoid the performance penalty of writing to disk for every password attempted and to avoid filling up the disk the loop only records data for every 100,000 attempts made by the process. If the process runs for too long it will be killed by the system scheduler so recording this data is important. Since only every 100,000 attempts are recorded some data may be lost, but a sufficient amount is still recorded to allow estimating the performance. If a process does find the solution it also records the discovered password in the log so manual verification can be made later.

Brute Force Password Generator

This generator creates passwords in memory based on alpha-numeric characters (0-9, A-Z, a-z) of length one to seven. When initialized from the process's main method it is given the current process's rank and the total number of processes. Each process calculates its appropriate start and end range of possible passwords based on the process rank and the number of processes. Only one password is stored in memory at a time which enables the generator to provide passwords without requiring a large amount of memory. Each call to the generator's nextPassword method provides the next successive password until the end of the range is reached at which time a special end signal is returned.

There are 62 possible alpha-numeric characters for each character in the password. Since a password may be from 1 to 7 characters in length this provides for $62^7 + 62^6 + 62^5 + 62^4 + 62^3 + 62^2 + 62^1 = 3{,}579{,}345{,}993{,}194$ possible passwords to attempt. Based on the number of processes and the process rank this can be divided up into ranges. One difficulty overcome was with specifying the start of a specific range for a process and translating that number into an actual password string. One method would be to simply start at the first possible password ('0') and increment by one until you roll into the desired start range. This however would have proven to be very slow especially for a process that may need the password at the one trillionth position for example. Instead an algorithm was devised that could quickly calculate the appropriate string password given a position (or number in the range) and the maximum possible number of passwords. The algorithm is as follows:

f(position) ➔

$$\text{position} == \text{factor}_1 * (\text{AlphabetLength})^{(\text{PasswordLength} - 1)} + \text{factor}_2 * (\text{AlphabetLength})^{(\text{PasswordLength} - 2)} + \text{factor}_3 * (\text{AlphabetLength})^{(\text{PasswordLength} - 3)} + \ldots + \text{factor}_{(\text{PasswordLength} - 1)} * (\text{AlphabetLength})^{(\text{PasswordLength} - (\text{PasswordLength} - 1))} + \text{factor}_{(\text{PasswordLength})} * (\text{AlphabetLength})^{(\text{PasswordLength} - (\text{PasswordLength}))}$$

where **AlphabetLength** is the number of possible combinations for a single character (62)

and **position** is a number from 1 to MaxPossiblePasswords(PasswordLength)

and **PasswordLength** is the length of the password for the given position calculated as follows:

PasswordLength = MaxPasswordsPossible(n) < position

for n = 1 to Max Password Length (7 characters)

such that n is the largest value which satisfies the above

then PasswordLength = n + 1

The goal is to find non-zero factors starting with factor_1 and going from left to right such that combining them with the above formula produces the value "position." Because multiple factor combinations are possible you want to start with making factor_1 the largest possible value that satisfies the above. The factors correspond to indexes in the alphabet character array set with factor_1 being the first character, factor_2 being the second, etc. A factor cannot be 0 because this could lead to a string value with invalid extra characters.

Example:

Suppose you have 3 possible characters: a, b, and c. Let the maximum password length be 3 characters. The 24th password in the range of 1 to 39 should be "bac." Assigning the alphabet array so that ALPHABET = {'a','b','c'} gives indexes of 0 = 'a', 1 = 'b', 2 = 'c' which indicates that our expected factors for a password of "bac" would be 1, 0, & 2. Using our algorithm to determine the factors results in:

$factor_1 * 3^2 + factor_2 * 3^1 + factor_3 * 3^0 == 24$

$factor_1 = 24 / 3^2 = 2$ with remainder 6

$factor_2 = 6 / 3^1 = 2$ with remainder 0

$factor_2 = 0 / 3^0 = 0$

However this results in the wrong factors of 2, 2, 0 or password "cca".

If instead we enforce that no factor may be 0 then our alphabet array index must be ALPHABET = {'\0', 'a', 'b', 'c'} which gives indexes of 0 = '\0', 1 = 'a', 2 = 'b', and 3 = 'c'. Note that the first item at index 0 is the null character and isn't used. It serves as a place holder. The number of possible password characters is still 3 and not 4. Anytime a factor would be zero we borrow one from the previous factor. If doing so would make the previous factor zero we have to set it to the maximum factor value (AlphabetLength) and borrow one from its previous factor. This may continue recursively as much as is needed.

Example:

$factor_1 * 3^2 + factor_2 * 3^1 + factor_3 * 3^0 == 24$

$factor_1 = 24 / 3^2 = 2$ with remainder 6

$factor_2 = 6 / 3^1 = 2$ with remainder 0.

$factor_3 = 0 / 3^0 = 0$. *Have to adjust.*

Borrow from $factor_2$ so that $factor_2 = 1$ with remaining now 3

Recalculate $factor_3 = 3 / 3^0 = 3$ with remainder 0.

This gives us factors 2, 1, and 3 which translate in our character array to "bac" our expected password. This allows for each process to quickly jump to the correct position based on its rank and the number of processes and produce a string password. Obtaining the next password is simply a matter of incrementing an internal counter variable and doing this same calculation again.

Dictionary Password Generator

In cryptanalysis and computer security, a dictionary attack is a technique for defeating a cipher or authentication mechanism by trying to determine its decryption key or passphrase by searching likely possibilities.

A dictionary attack uses a targeted technique of successively trying all the words in an exhaustive list of pre-arranged values. In contrast with a brute force attack where a large proportion key space is searched systematically a dictionary attack tries only those possibilities which are most likely to succeed, typically derived from a list of words like a dictionary (hence the phrase dictionary attack) or a bible etc. Generally, dictionary attacks succeed because people have a tendency to choose passwords like the following: short (7 characters or fewer), single words found in dictionaries, or simple and easily predicted variations on words such as appending a digit.

Building larger dictionaries

John the Ripper is a free password cracking software tool. Initially developed for the UNIX operating system, it currently runs on fifteen different platforms (11 architecture-specific flavors of UNIX, DOS, Win32, BeOS, and OpenVMS). It is one of the most popular password

testing/breaking programs as it combines a number of password crackers into one package, auto detects password hash types, and includes a customizable cracker.

One of the modes John can use is the dictionary attack. It takes text string samples, usually from a file called a *wordlist* containing words found in a dictionary, encrypting them in the same format as the password being examined (including both the encryption algorithm and key), and comparing the output to the encrypted string. It can also perform a variety of alterations to the dictionary words and try these. Many of these alterations are also used in John's single attack mode which modifies an associated plaintext (such as a username with an encrypted password) and checks the variations against the encrypted hashes.

The command used to make permutations and combinations of words in an existing wordlist or dictionary was

```
john --wordlist=all.lst --rules --stdout | unique mangled.lst
```

Here, all.lst was the existing dictionary (32MB with 3,160,121 words), and the mangled.lst (about 800MB with 85,472,799 words) was the permuted version of it by applying all the rules. The utility ‘unique’ was used to eliminate all the duplicate passwords.

Approach

The initialization of the dictionary involves three steps. All these steps are executed by all the processors:

Step 1: Counting the total number of words in the dictionary

The initial step involves counting the number of words in the existing dictionary. This is done by all the processes. As shown in figure 2 below, the counting of words is done serially by all the processors, and hence it takes n iterations where n is the number of words in the dictionary file.

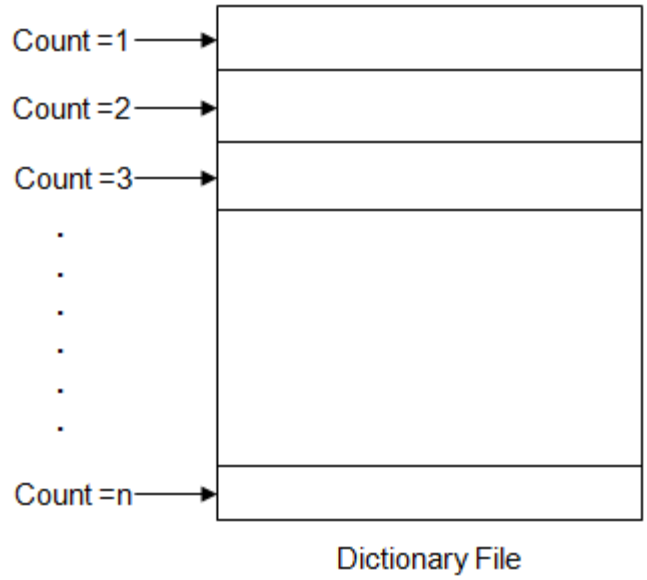


Figure 2. Every process counts the total number of words in the dictionary.

Step2: Calculating the displacement array

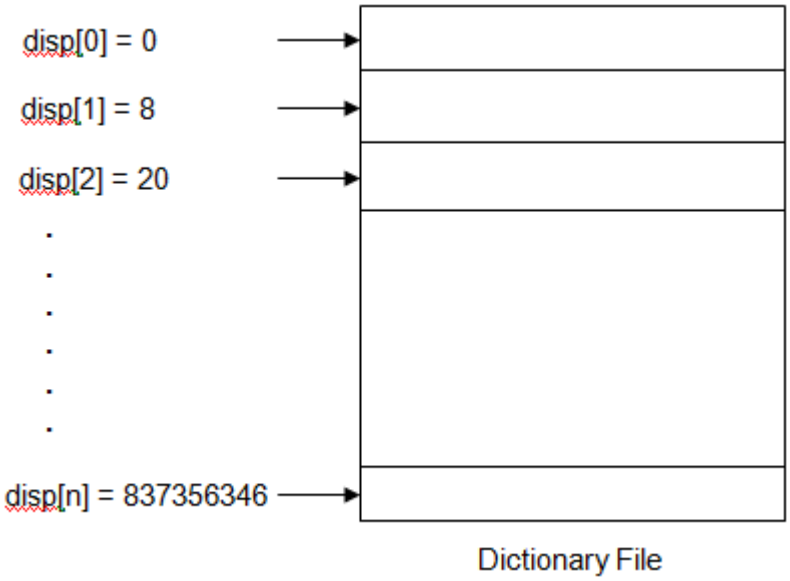


Figure 3. Every process calculates offsets for all the words in displacement array disp[].

The displacement or the offset for all the words is needed so that different processors can index into different parts of the dictionary. As shown in figure 3 above, this step is also done by all the processors in exactly the same way, and it adds another n iterations.

Steps 1 and 2 are a bottle neck in the initialization process because we have designed the dictionary initialization process with the assumption of not knowing the number of words beforehand.

Step 3: Indexing into the Table

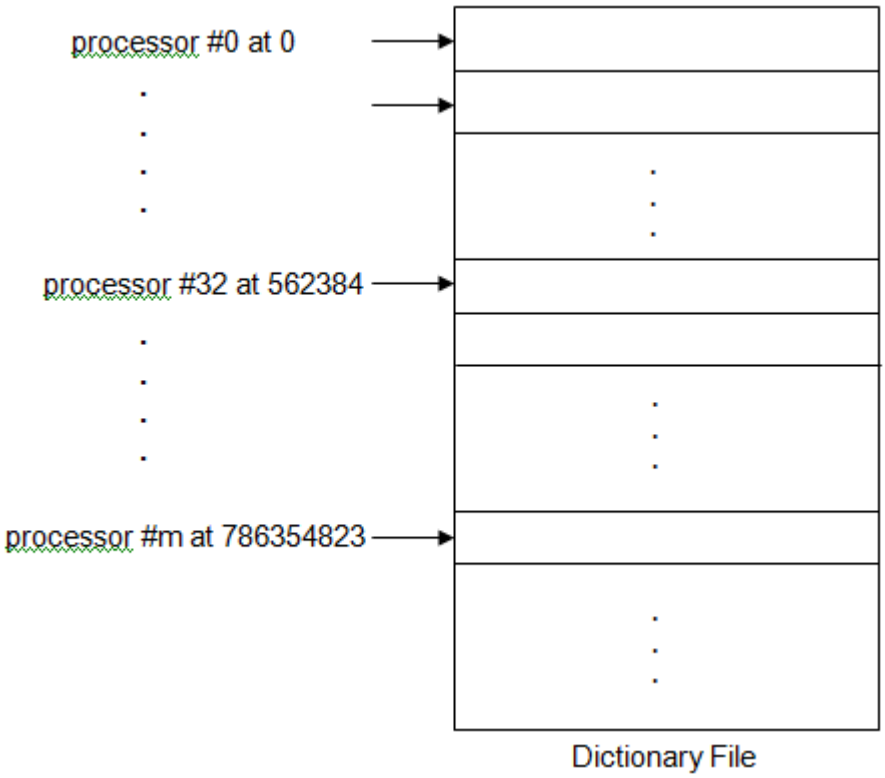


Figure 4. Sample index locations for the m processors

The formula used to index into the dictionary file requires three things.

- The number of words each process needs to look for (i.e. per process word count)
    - i.e. (approximately n/m) where n is the total number of words and m is the number of processors.
- The rank of the process.
- The displacement array calculated in step 2.

At the end of this step, as shown in figure 4 above, the process #32 indexes at location 562,384 byte offset from the start of the dictionary file. It also has calculated how many words it has to attempt. This step thus completes the process of initialization of the dictionary where every process is indexed at the appropriate places in the dictionary file with their per process word counts.

Balancing load

The amount of work is distributed evenly among all the processes. Example: If we have the number of words in a dictionary as n = 103 and the number of processors m = 10 then initially the per process word count is 103 / 10 = 10 (by integer division). Thus

- processor with rank 0: 10
- processor with rank 1: 10
- processor with rank 2: 10
- processor with rank (3-9): 10

Then the remaining words (103 % 10 = 3) are given one each to all the processes with rank < 3 giving us the following distribution:

- process with rank 0: 11 words
- process with rank 1: 11 words
- process with rank 2: 11 words
- process with rank (3-9): 10 words

Thus the amount of words is distributed as evenly as possible amongst all the processors.

Decrypt Engine

In order to crack a zip file, i.e. find out the password used during the encryption process of the zip file, we first need to understand its format, parse the zip file to collect all the information, and then work on the available information.

Zip file format:

Here we will look at the format of the zip file according to the specification used by WinZip[1]

HEADER
FILE NAME
EXTRA FIELD
SALT
PASSWORD VERIFIER
ENCRYPTED FILE DATA
AUTHENTICATION CODE (MAC)

Figure 5: Zip File format

1. Header:

This is a 30 byte field. It contains the information of the zip file and the process through which it was zipped. It contains information like the header signature, the version, compression method used, the time and date when the file was last modified, and the size of the compressed and uncompressed data. Also, it contains the length of the next two fields: File Name and Extra field.

[1] http://www.winzip.com/aes_info.htm

2. File Name:

The length of this field is extracted from the header. It contains the name of the original file that was zipped.

3. Extra field:

This is an 11 byte field containing information about the size of the data field and also the mode used. There are 3 modes used for encrypting the data corresponding to AES-128, AES-192, and AES-256. The mode depends on the size of the password provided for encrypting and defines the encryption strength. Mode is required while decrypting the data.

4. Data Field:

This field contains the rest of the file including salt, password verifier, encrypted data, and the message authentication code.

4.1 Salt:

The "salt" or "salt value" is a random or pseudo-random sequence of bytes that is combined with the encryption password to create encryption and authentication keys. The salt is generated by the encrypting application and is stored unencrypted with the file data. The addition of salt values to passwords provides a number of security benefits and makes dictionary attacks based on pre-computed keys much more difficult. The size of the salt depends on the length of encryption key as outlined in this table:

Key size	**Salt size**
128 bits	8 bytes
192 bits	12 bytes
256 bits	16 bytes

Table 1. Salt lengths based on AES encryption strength

4.2 Password Verifier:

This two-byte value is produced as part of the process that derives the encryption and decryption keys from the password. When encrypting a verification value is derived from the encryption password and stored with the encrypted file. Before decrypting, a verification value can be derived from the decryption password and compared to the value stored with the file serving as a quick check that will detect most, but not all, incorrect passwords. There is a 1 in 65,536 chance that an incorrect password will yield a matching verification value; therefore, a matching verification value cannot be absolutely relied on to indicate a correct password but in the case of our project can be used to eliminate majority of the incorrect passwords. This value is stored unencrypted in the file.

4.3 Encrypted File Data:

Encryption is applied only to the content of the file. It is performed after compression and not to any other associated data. The file data is encrypted byte-for-byte using the AES encryption algorithm operating in "CTR" mode, which means that the lengths of the compressed data and the compressed encrypted data are the same.

Here the data is divided into blocks of 16 bytes each for either encryption or decryption. As we are looking at resolving the passwords in this project we do not consider the compression process.

4.4 Message Authentication code (MAC):

Authentication provides a high quality check that the contents of an encrypted file have not been inadvertently changed or deliberately tampered with since they were first encrypted. This value for the authentication code can be used to confirm the correctness of the password.

The authentication code is derived from the output of the decryption process and is stored unencrypted. It is byte-aligned and immediately follows the last byte of encrypted data.

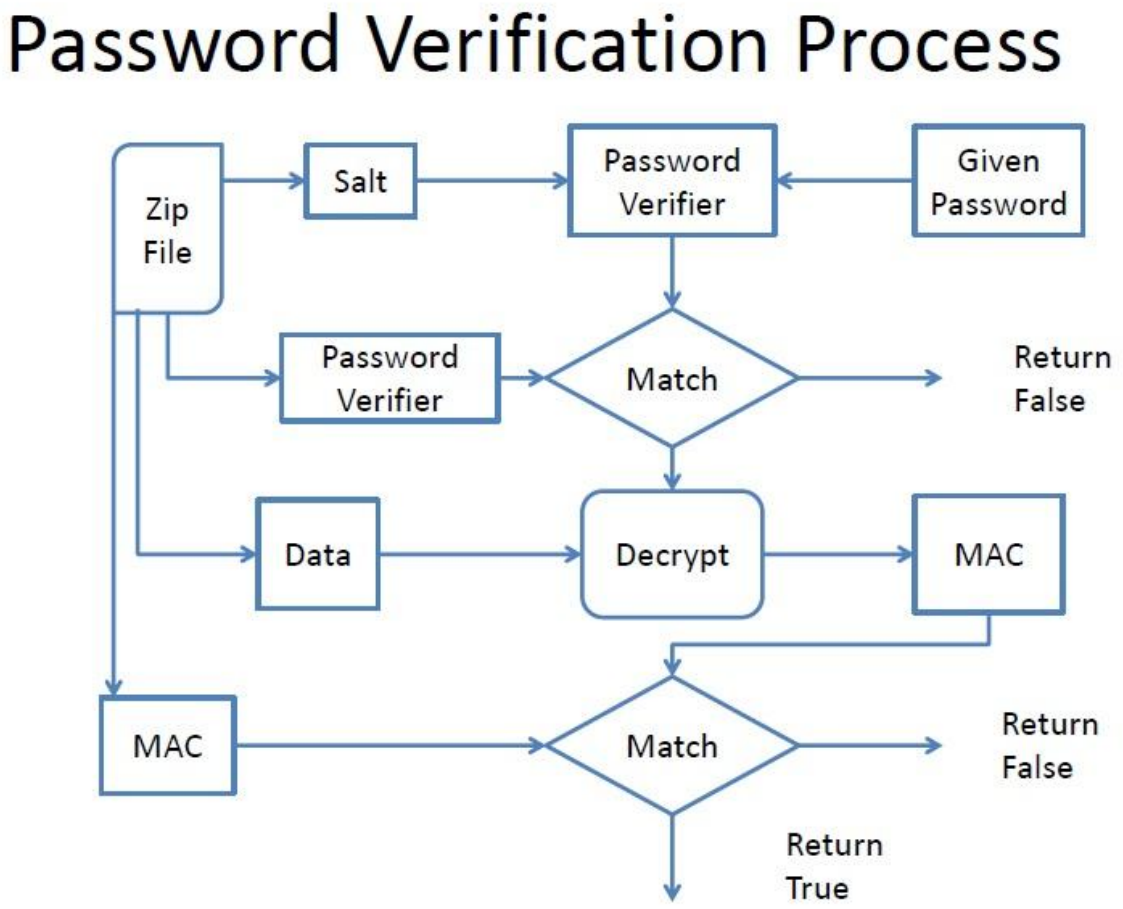


Figure 6: Flowchart of Password verification process

The Process:

Input:

Input to the password verification process is the zip file for which the password is to be recovered, and the current password which we need to verify.

Outline:

The process consists of two operations, namely the quick 2 byte check and decryption. As the number of passwords given to the password verifier is huge, we need a mechanism to reduce

the possibly correct passwords. For this we use the two byte quick password verifier check. There is a 1 in 65,536 chances that an incorrect password will yield a matching verification value. Thus for those passwords which match the verification value we check the authentication value and eliminate all the incorrect passwords, finding the one correct password if present.

We used Dr. Brian Gladman's AES code to decrypt the zip file. It is a C library for AES encryption and decryption. It is also used by "WinZip"[2]. The following is the step by step explanation of the flowchart:

Step 1: Initially we extract the salt from the zip file.

Step 2: With the given password and extracted salt, we call the fcrypt_init() function from Gladman's code to get the password verifier. The function initializes the decryption stream and returns the password verifier.

Step 3: Then the password verifier stored in the zip file is extracted.

Step 4: The two password verifiers from step 1 and 2 are compared to eliminate most of the incorrect passwords.

Step 5: If they do not match return false. This is where majority of the incorrect passwords are detected.

Step 6: Even if the password verifiers match, there is a possibility that the given password is incorrect. Thus we need to confirm the correctness of the password for which we match the message authentication codes. An incorrect password can pass the quick two byte check because of the very short length of the password verifier (2 bytes). There are only 65,536 different password verifiers, but the number of passwords to be tried is much larger than this.

[2] http://www.gladman.me.uk/cryptography_technology/fileencrypt/

Step 7: Here, the encrypted data from the zip file is extracted, and it is decrypted using fcrypt_decrypt() function from Gladman's code. This process requires dividing the zip file data into blocks of 1024 bytes. After decryption, the fcrypt_end() function is called, which returns the message authentication code.

Step 8: The message authentication code, obtained during encrypting the file, is present in the zip file which is now extracted.

Step 9: The two MACs from step 7 and 8 are now compared.

Step 10: If the MACs do not match then the password was incorrect, i.e. it matched the password verifier but could not decrypt the file correctly. Return false.

Step 11: If the authentication MACs matched, we have found the password! Return successfully from the function.

The Code:

As seen from the above steps we use 3 functions from Gladman's code. They are fcyrpt_init(), fcrypt_decrypt and fcrypt_end()[3]. Here the generator is used rather like an I/O stream: it is opened (initialized), used, and finally closed. The password verifier is returned when the stream is opened, and the authentication code is returned when the stream is closed.

1. Initialize the "stream" for decryption and obtain the password verification value:

```
fcrypt_ctx zctx;          // the encryption context

int rc = fcrypt_init(

  KeySize,       // extra data value indicating key size
```

[3] http://www.winzip.com/aes_tips.htm

```
  pszPassword,            // the password
  strlen(pszPassword), // number of bytes in password
  achSALT,      // the salt
  achPswdVerifier,   // on return contains password verifier
  &zctx);          // encryption context
```

The return value is 0 if the initialization was successful. Non-zero values indicate errors. The function returns the password verification value in *achPswdVerifier*, which must be a 2-byte buffer. The initialized encryption context (*zctx*) is used as a parameter to the decryption functions. Therefore, its state must be maintained until the "stream" is closed.

2. Decrypt the data:

We use the following function to decrypt the data.

```
fcrypt_decrypt(
  pchData, // pointer to the data to decrypt
  cb,   // how many bytes to decrypt
  &zctx); // decryption context
```

We call the decrypt function multiple times, passing in successive chunks of data in the buffer. For AE-1 and AE-2 compatibility, the buffer size is a multiple of 16 bytes except for the last buffer, which may be smaller.

3. Close the "stream" and obtain the authentication code:

When decryption is complete, the "stream" is closed as follows:

```
int rc = fcrypt_end(
  achMAC, // on return contains the authentication code
  &zctx); // encryption context
```

The return value is the size of the authentication code, which will always be 10 for AE-1 and AE-2. The authentication code itself is returned in your buffer at *achMAC*, which is an array of *char*, sized to hold at least 10 characters.

Possible speedup:

For each run of the code we supply it with a zip file and a large number of passwords through brute-force or dictionary attack. For efficiency, the file handling operations are reduced by opening the zip file only once, extracting all the required information, and storing it into a data structure. With each try of the password, this data structure is used instead of opening the file again.

Also, we avoid decrypting the file for most of the passwords by checking the 2 byte password verifier value. The file is decrypted only for the passwords which pass the test. This reduces the computation time a lot as file decryption is a time consuming process.

Now the major bottle neck remaining is the decryption process when the password has passed the two byte check. Currently the implementation of this process is serial. If we were to parallelize the decryption process on GPU machines as suggested by Deguang Le et. al. in their paper "Parallel AES algorithm for fast encryption on GPU" we would expect to further improve the running time of the code.

Testing Methodology

Input test data was created for both password generator types of brute and dictionary. Each test file was a zip file created and encrypted using the tool 7-zip[4] which uses the AES-256 encryption cipher according to the zip file format specification[5]. Each zip file contained a single plain text document and was kept small enough in size to fit easily into memory. For each password generator type a zip file for the first, middle, and last passwords in the range was created. In addition both types had a zip file encrypted with a password that was known to not be in the range of possible passwords. This test data allowed us to collect performance data on how many password attempts could be made using a given number of processes as well as the time to find a solution or exhaust all possibilities.

We ran our program on two different supercomputer clusters. NCAR's Frost which has 4,096 nodes total with each node having a 700MHz dual-core CPU and 512MB of RAM, and on NCAR's Janus which has 1,368 nodes with each node having two 2.8 GHz six-core CPUs and 2GB of RAM. For each test file we ran the program on both systems with varying processor counts of 128, 1024, 2048, and 4096. As mentioned in the framework section each process records its own time stamped log of the number of attempts it made and whether it found the solution or not. We allowed each test to have the maximum wall-time of 24 hours on both

[4] http://www.7-zip.org/ Igor Pavlov
[5] http://www.pkware.com/documents/casestudies/APPNOTE.TXT

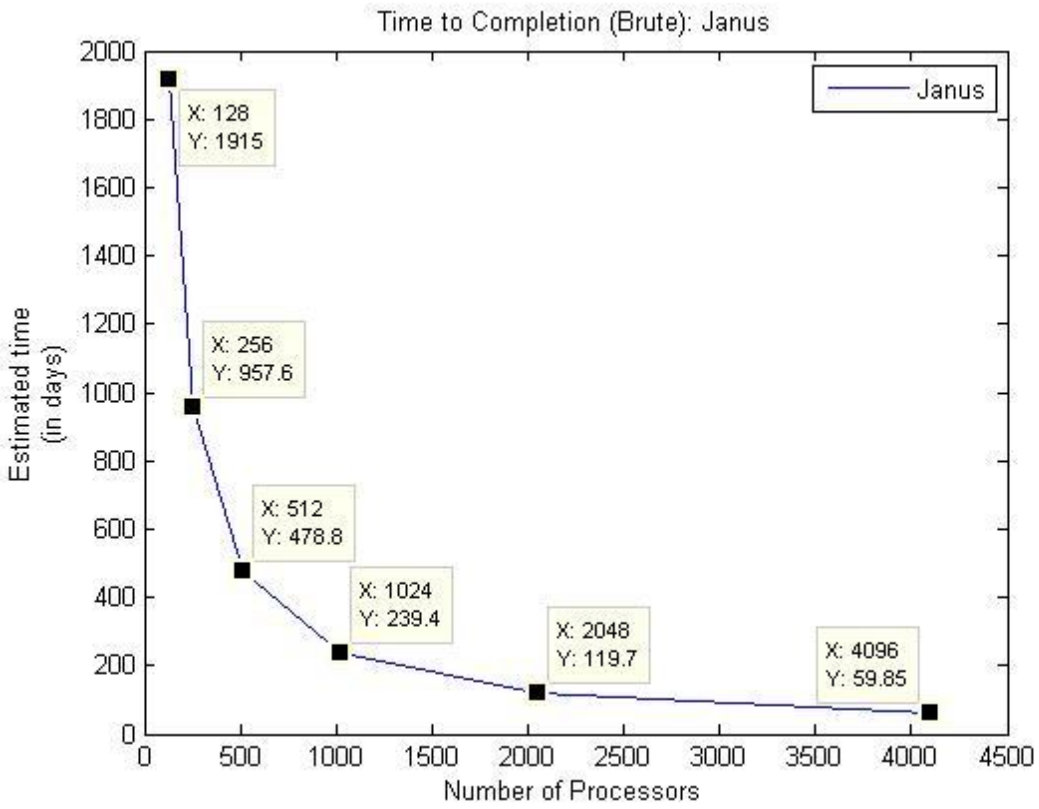
systems. Some tests ran long enough to reach their wall-time and were killed by the system, but since each process logged its progress as it went we were still able to get useful data for this 24 hour period to allow us to estimate the amount of time it would have taken to completely finish.

Performance Testing

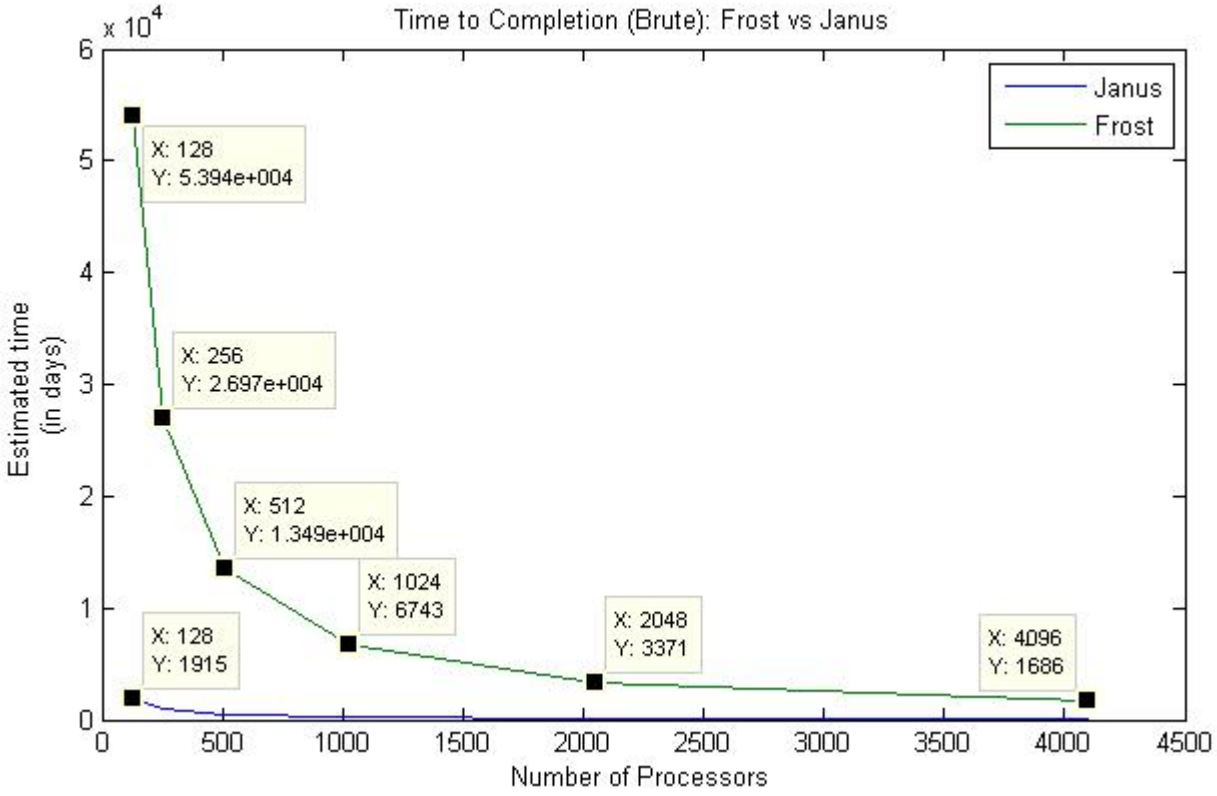
To test performance on Frost and Janus, runs were made for processor counts of 128, 1024, 2048, and 4096. The results were interpolated for processor counts of 256 and 512. Since the brute force run has a very large search space, the runs did not complete when the password was in the middle, in the end, or not in the search space. Some of the dictionary runs could also not complete for a processor count of 128 on the Frost system.

Results

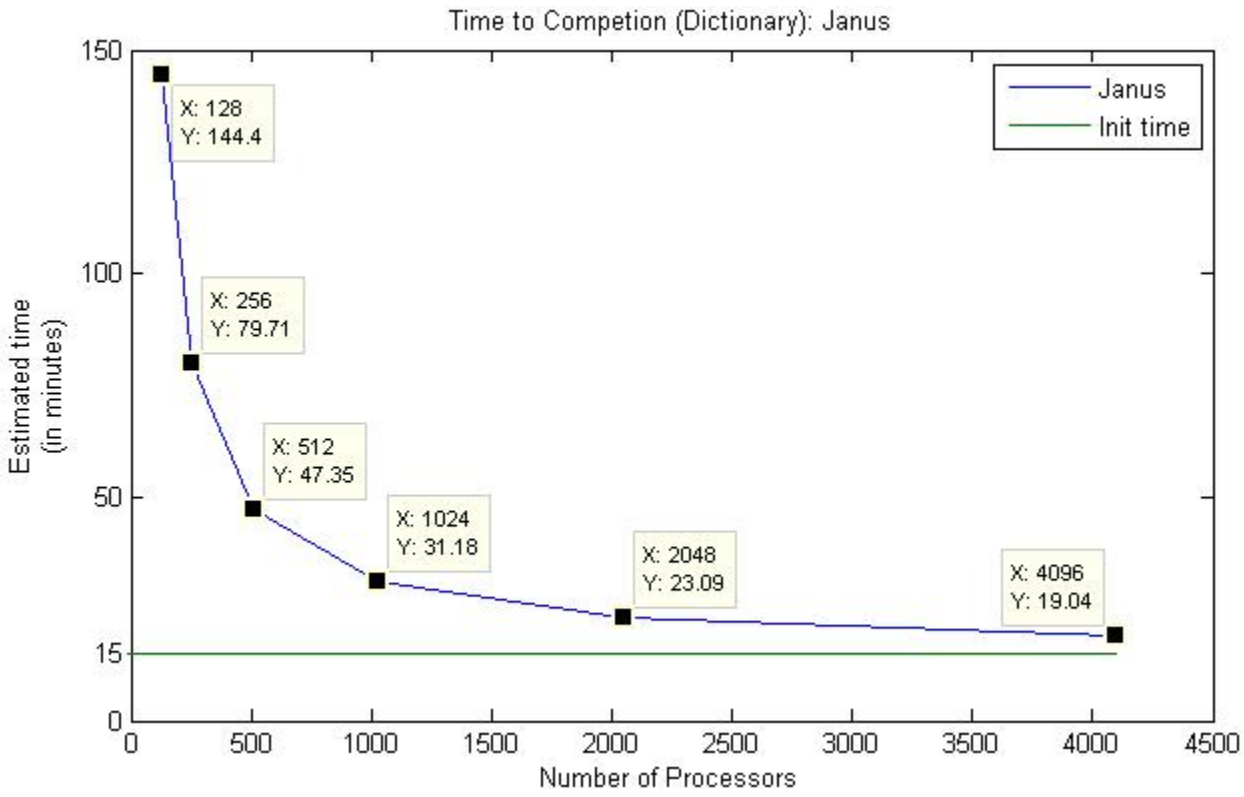
1) Time to complete search space with BRUTE mode (Janus)



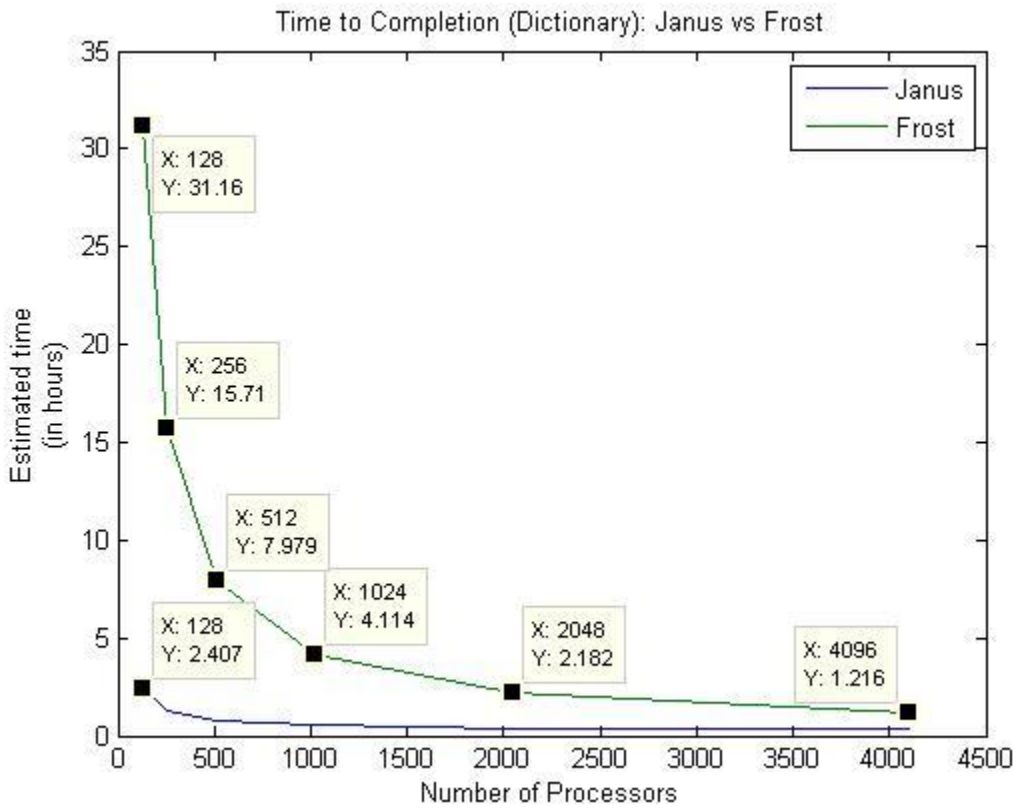
2) Time to complete search space with BRUTE mode (Frost vs Janus)



3) Time to complete search space with DICTIONARY mode (Janus)



4) Time to complete search space with DICTIONARY mode (Frost vs Janus)



Conclusions

We were able to derive the following formula for estimating the time it would take to search a given key space with a given number of processes:

Let **n** be the number of words in the search space.

Let **m** be the number of processes.

Let **A** be the performance of the supercomputer as measured as the average number of attempts per second by one process

$$f = (n) / (A * m)$$

The resulting value is the number of seconds to search the entire key space.

Search space:

- Brute, n = 3,579,345,993,194
- Dictionary, n = 85,472,799

Average number of attempts per second by one process::

Method	**Frost**	**Janus**
Brute	6	169
Dictionary	6	86

The best performance was obtained on the Janus system with a 4,096 processor count with the following major observations:

- Max throughput
  - Brute = 172 passwords / second
  - Dictionary = 228 passwords / second
- Average throughput
  - Brute = 169 passwords / second
  - Dictionary = 86 passwords / second
- Brute method complete key space search time (projected)
  - 7 alphanumeric chars = 60 days
  - 8 alphanumeric chars = 9.9 years
  - 10 alphanumeric chars = 38,395 years
- Dictionary method complete key space search time (projected)
  - 1 billion words = 47.3 minutes with 4096 processors
  - 100 billion words = 78.85 hours with 4096 processors

Our test results show that the possibility of recovering a zip file archive password that either exists in a dictionary of billions of words or is 7 alphanumeric characters or less is quite feasible to do in a reasonable amount of time. This emphasizes the importance of choosing good passwords to protect sensitive data since computing power has advanced and will continue to advance to levels that allow guessing the password to be possible. As our formula shows passwords of length eight or more would require significant time to find via brute force. Adding

to the complexity can also defeat the possibility of a dictionary attack successfully finding the password.

Future Follow-up

Currently the implementation of the decryption process is serial. We can parallelize the decryption process on GPU machines as suggested by Deguang Le et. al. in their paper "Parallel AES algorithm for fast encryption on GPU."

The potential bottleneck in the dictionary mode with parsing words and offsets could be minimized by dividing the dictionary file before hand into smaller pieces so that each processor gets one entire file or by calculating the number of words and displacements offline and passing the information as arguments to the program. Also, it would be interesting to see if the rate of password look-ups for the dictionary method increases with processors reading from different dictionary files rather than just one dictionary file as in our case.

References

Agrawal, Neelam, Rodney Beede, and Yogesh Virkar. PZAPR Project Page. Google Code. Web. 4 May 2011. < http://code.google.com/p/pzapr/>

"APPNOTE.TXT - .zip File Format Specification Version: 6.3.2." *Support Zip AppNote*. PKWARE Inc. 2007. Web. 12 March 2011 <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>

Bishop, Matt & Daniel V.Klein. "Improving System Security via Proactive Password Checking." *Computers and Security* 14(3). May/June 1995. Print.

Front Range Computing Consortium. Janus ATP v1.0 documentation. Web. 29 April 2011. <http://ncar.janus.rc.colorado.edu/>

Le Deguang, Jinyi Chang, Xingdou Gou, Ankang Zhang, and Conglan Lu. *Parallel AES Algorithm for Fast Data Encryption on GPU.* Changshu: Changshu Institute of Technology, China, 2010.

Web. 14 March 2011.
<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5486259>

“Password Recovery Attacks." Passware Inc. n.d.
Web. 15 March 2011.
<http://www.lostpassword.com/attacks.htm>

United States. National Institute of Standards and Technology. *Announcing the ADVANCED ENCRYPTION STANDARD (AES).* Maryland: NIST. 2001.
Web. 15 March 2011.
<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>