Implementation of a Secure Distributed Storage System

Snehil Suresh Wakchaure
Simrit Kaur Arora

ECE646 Fall 2012 Project Specifications
Department of Electrical & Computer Engineering
George Mason University

October 6, 2012

swakchau@masonlive.gmu.edu
sarora9@masonlive.gmu.edu

Introduction:

Secure distributed data storage techniques are gaining importance with the increase of data centres, mobile devices & the need to save encrypted data in a distributed environment. In this project, we intend to do performance evaluation of selected open source encryption algorithms, secret sharing schemes and erasure coding algorithms. We also, intend to implement AONT (All-or-nothing transform) and merge this implementation in our implementation of the secure distributed storage system. Next, we intend to develop the entire distributed storage system for PC and mobile (Android) platforms.

The resulting system supports two important properties:

1. Data can be recovered if & only if some minimum number of data fragments is recovered i.e. if no. of fragments recovered are less than the minimum number required, then no data is recovered, and

2. Sensitive data remains protected even after a small number of data fragments are compromised.

Development Tools:

PC:
OS: Linux Ubuntu 3.2.0-23 generic 64-bit
Language: C/C++, Java for Android
Compiler: GCC/G++ GNU compiler
Debugger: Valgrind [20] for debugging/system profiling or GDB debugger
Hardware: 3.2 GHz Intel I3 350M multithreaded processor with 8GB of RAM and 3MB of L3 cache

Android:
Language: Java
Execution Platform: Nexus 7
Specifications:
Internal Memory: 16 GB storage, 1GB RAM
OS: Android, v4.1(Jelly Bean)
Chipset: Nvidia Tegra 3
**CPU:** Quad-core 1.3 GHz Cortex-A9

**Additional Libraries:**

The following 3 types of transformations shall be used to realize the system:
(a) Encryption  
(b) Secret Sharing Schemes  
(c) Erasure coding techniques  
(d) All-or-Nothing Transform

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Library</th>
<th>Language</th>
<th>Type of Transformation</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>libgfshare</td>
<td>C</td>
<td>secret key sharing</td>
<td>Shamir's scheme</td>
</tr>
<tr>
<td>2</td>
<td>cryto++</td>
<td>C/C++</td>
<td>Encryption</td>
<td>AES</td>
</tr>
<tr>
<td></td>
<td>cryto++</td>
<td>C/C++</td>
<td>secret key sharing</td>
<td>Shamir's scheme</td>
</tr>
<tr>
<td></td>
<td>cryto++</td>
<td>C/C++</td>
<td>erasure coding</td>
<td>Rabin's IDA</td>
</tr>
<tr>
<td>3</td>
<td>ssss</td>
<td>C</td>
<td>secret key sharing</td>
<td>Shamir's scheme</td>
</tr>
<tr>
<td></td>
<td>LUBY</td>
<td>C</td>
<td>Erasure coding</td>
<td>Cauchy Reed Solomon</td>
</tr>
<tr>
<td></td>
<td>SCHIFRA</td>
<td>C++</td>
<td>Erasure coding</td>
<td>Standard Reed Solomon</td>
</tr>
<tr>
<td>6</td>
<td>ZOOKO</td>
<td>C</td>
<td>Erasure coding</td>
<td>Standard Reed Solomon</td>
</tr>
<tr>
<td>7</td>
<td>JERASURE</td>
<td>C/C++</td>
<td>Erasure coding</td>
<td>Standard &amp; Cauchy Reed Solomon, Minimal density RAID-6</td>
</tr>
<tr>
<td>8</td>
<td>CLEVERSAFE</td>
<td>C</td>
<td>Erasure coding</td>
<td>Cauchy Reed Solomon</td>
</tr>
<tr>
<td>9</td>
<td>EVEN/ODD RDP</td>
<td>C/C++</td>
<td>Erasure coding</td>
<td>Parity array</td>
</tr>
</tbody>
</table>

**Assumptions & Restrictions:**

**Memory assumptions for PC:**

**Tools:** Valgrind [20] may be used for memory profiling.

**Heap:** Dynamic memory is allocated on the heap the available memory for the process can be measured programatically. I am currently using Linux VM and using 'ulimit -a' returns the memory
available as 'unlimited'. So, the code in Appendix A: Code A1 was used calculate the approximate available memory per process in my system and gives the following result.

\[ \text{Max Usable Memory per process} = 6396313600 \text{ Bytes} = 6GB \text{ approximately} \]

**Stack:** Using `ulimit -a` gives the stack size as 8192Kbytes for the current configuration used. Measures have to be taken by setting the appropriate limits in case of a Stack overflow & that the hard limit is not violated by the executing process.

Dynamic memory allocation shall be preferred as a preventative measure against Stack overflow wherever possible.

**For android devices:**
The size of your APK file is limited to 50MB to ensure secure on-device storage, but expansion files can be attached to the APK. Each app can have two expansion files, each one up to 2GB, in whatever format we choose.

**Experimental setup**

The experiment would be carried out in stages parallel for PC based & Android based implementations:

**Stage 1: Performance evaluation of Erasure coding, Secret key sharing & implementation of AONT schemes**
The experimental setup for Stage 1 would comprise of a C/C++ code to measure the execution time of the encoding/decoding process for performance evaluation of the available algorithms.

**Stage 2: C/C++/Java implementation of the Secure Distributed Storage System**
The proposed experimental setup for Stage 2 is as shown in the figures below:
2(a)
The data for distributed storage is encrypted using AES & secret key sharing algorithm, then AONT transformed & finally erasure coded to produce n data fragments out of which minimum t are required to regenerate the original data (Erasure code). If less than t fragments are available then no part of the data can be decrypted (AONT).

2(b)
The node in the network generating the data fragments communicates with the other active nodes & distributes the fragments (1 per node) generated.
2(c)
The node that wants to regenerate the original data, first checks how many nodes are active in the network. It can regenerate the data only if the minimum number (t) nodes are active. It then collects the data fragment (>=t) from the active nodes.

2(d)
After getting the required number of data fragments (>=t), erasure decoding is done. Then, inverse of the all-or-nothing transform & finally decryption of data using DES & Secret sharing algorithm.
Files, Testing & Verification:

**Input files:**
For Encoder
- Configuration file (text file containing information about the data size, number of fragments into which data is to divided & the minimum no. of these fragments required to reconstruct the original data)
- Test input data file
For Decoder
- Fragmented input data files
- Configuration file (text file)

**Output files:**
For Encoder
- Fragmented data files
- Output report
For Decoder
- Re-constructed input data file
- Output report

**Testing: Source vectors (The input data file)**
Testing would be in terms of
1. Efficiency: given the size of the input file and the number of nodes among which the data is to be divided, the time taken to implement the design.
2. Analysing the integrity of data of the file which is stored in a distributed way.
3. Exact regeneration of the input file by collecting data fragments from various nodes.

**Test Cases:**
1. Read the data from RAM. Encrypt the data using the AES encryption. Apply AONT & divide it into n fragments using erasure code (such that at least t fragments are required to reconstruct the data. Any less than t fragments do not reveal any data). Recollect (≥ t) fragments at a different node. Apply erasure decode, inverse of AONT & decryption scheme to regenerate the original message & store in RAM of the data requesting node.
**Test Case 2:**
Same as test case 1. Except this time the data to be encoded is picked up from Disk at the originating node & stored into the disk of the data requesting node.

**Analysis for an optimum value of (n,t):**

n: Total number of fragments into which data is divided.

<table>
<thead>
<tr>
<th>n</th>
<th>Total number of fragments into which data is divided.</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Minimum number of fragments required to recover the data.</td>
</tr>
</tbody>
</table>

Below shows a graph with respect to a case where n varies from 6 to 12 & t varies from 2 to 4. If we increase the value of n keeping the value of t constant, the execution time increases. Also, there will be a variation when the value of n is kept constant but that of t is changes. Hence, such a value of (n,t) is required that satisfies the security criteria & also takes the minimum possible encoding/decoding time.

*Graph showing the relationship between data size (n) and execution time for coding/decoding.*

\[ n \leq 12 \quad \text{and} \quad t \leq 6 \]
\[ n \in (6, 12) \quad \text{and} \quad t \in (2, 4) \]
\[ n_1 > n_2 \]
List and initial analysis of similar programs reported earlier in the literature or on the web:

<table>
<thead>
<tr>
<th>Feature</th>
<th>MDFS</th>
<th>Tahoe-LAFS</th>
<th>Unisys Stealth</th>
<th>GFS &amp; Bigtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypted</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Erasure coded</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Complete replication</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Pre-share key</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Relies on external authentication</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scales to large sizes</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Relies on external infrastructure</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2.1: Distributed Storage Comparison

Time Schedule:

Oct. 15-17: Performance testing of the available codes
**Most efficient will be used for the design
Oct. 29-31: Implementation of AONT & the entire scheme.

Tentative Table of contents:

1. Introduction
2. Background
   2.1 Distributed Storage Basics
   2.2 Advanced Encryption Standard (AES)
   2.3 Erasure Codes
   2.4 Shamir’s Secret Sharing Algorithm
   2.5 All-or-Nothing Transform
   2.6 Putting it all together
3. Concept and Design
4. Experimental Results
   4.1 Equipment Used
   4.2 Device Limitations
   4.3 Application Details
   4.4 Implementation Details
   4.5 Experimental Setup
   4.6 Experimental Results
5. Conclusion

List of References
References


[12] Victor Boyko On the Security Properties of OEAP as an All-or-Nothing Transform

[13] Ronald L. Rivest All-or-Nothing Encryption and the Package Transform

[14] D. R. Stinson Something About All or Nothing (Transforms)


APPENDIX A

**Code A1: Code to check maximum memory allocated to a process**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(){
    size_t maxmemperprocess=0; // Max. memory avlbl. per process in bytes
    size_t fiftyMB=52428800; // 50MB in Bytes
    void *memPtr= NULL; // Pointer to test memory block
    do{
        if(memPtr!= NULL){
            printf("Testing Memory= %zi\n",maxmemperprocess); //Print current memory size
            memset(memPtr,0,maxmemperprocess); //Reset memory
            free(memPtr); // Free memory block
        }
        maxmemperprocess+=fiftyMB; //Increment memory block by 50MB
        memPtr=malloc(maxmemperprocess); //Allocate current block size of memory
    }while(memPtr!= NULL); // Exit loop when max. memory allocation reached
    printf("Max Usable Memory per process aproximately= %zi Bytes\n",maxmemperprocess-fiftyMB);
    return 0;
}
```

// -------------------- OUTPUT -----------------------------

// Testing Memory= 52428800
// Testing Memory= 104857600
// Testing Memory= 157286400
// .
// .
// .
// Testing Memory= 6239027200
// Testing Memory= 6291456000
// Testing Memory= 6343884800
// Testing Memory= 6396313600
// Max Usable Memory per process aproximately= 6396313600 Bytes
//-------------------------------------------------------------------------------------