CSCE-110 Fall 2019: Term Project

The Safe Box

Phase 2

Better Performance and More Concealment

Assigned: Tuesday, November 19, 2019

Due Date: Tuesday, November 26, 2019

Overview

In this phase your are required to apply concurrency support using C++11 threads to your safe box implementation. Basically, you will need to create a separate thread to handle each shred at a time, through which the AES algorithm will be run in parallel on different blocks within different shreds. This will enhance the execution speed of the import and the export functionalities of your Safe Box. Moreover, instead of using the predictable round-robin dispatching of blocks over shreds, you will use an unpredictable random lottery-based dispatching approach to conceal the order of blocks within shreds. You will be able to achieve 2 targets with the same bullet; more performance and more concealment through multi-threading.

Details

In this phase, you will amend your initial design and implementation of Safe Box to support concurrency. You will use C++11 threads to implement a multi-threaded version of the Safe Box to provide the import and export functionalities. You will precisely create a thread for each shred. Typically, we have two cases here, import and export, which both will utilize multi-threading. In the import case, you will have to create a thread for each shred. A single file spooler will continue to be used by all threads, and hence will need to be carefully synchronized among threads. Each thread will have a loop that preform the following:

1. Request a block from the file spooler.
2. Encrypt the block.
3. Store the encrypted block into the corresponding shred file.

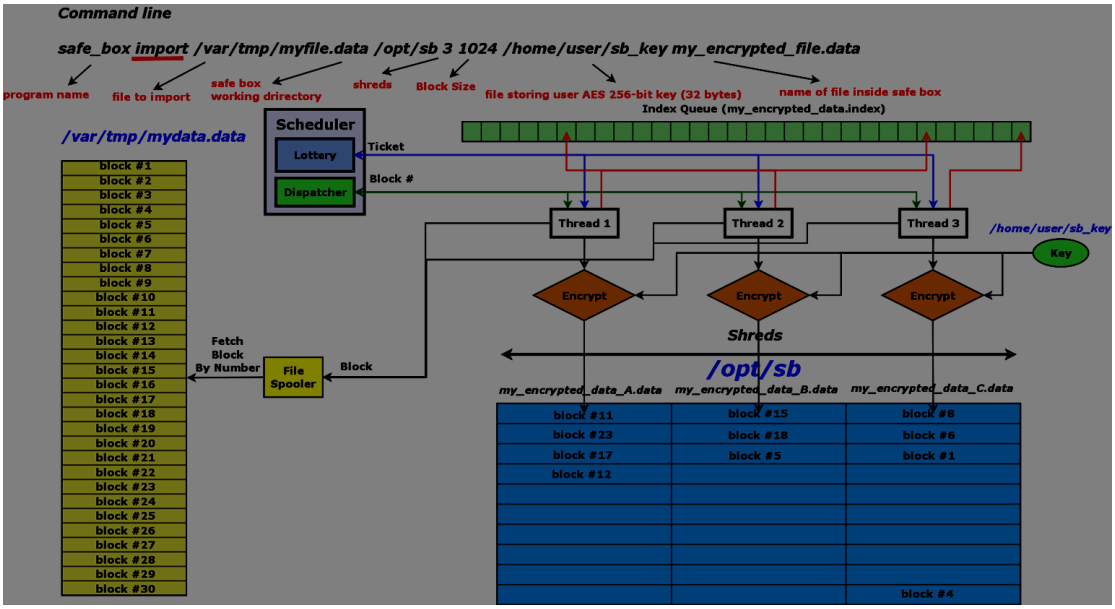
It is very important to highlight that we cannot assume any order of the threads block requests from the file spooler as this precisely depends on the operating

system preemption decisions, as well as the thread library runtime scheduling events, and hence the order of the blocks across the different threads cannot be assumed to be round-robin. Essentially, you will need to maintain some information about where each block resides; precisely which shred. For that purpose, each imported file will need to have an index in the form of a QUEUE, built during the import and to be used in the case of export; we will call it the **Index Queue**. Each element in the Index Queue corresponds to a block and its order is corresponding to the order of the block in the original file. Each element in the Index Queue should be implemented as a C-Struct that contains which shred the block is stored in, its location within the shred, and its location in the original file. It is pretty obvious that this queue need to be synchronized for concurrent access by different threads.

Using this approach, the distribution of the blocks among shreds will be semi round-robin! meaning that it is based on the speed of compression by different threads and how much time slice each thread is assigned by the OS and/or the thread library. It will be pretty difficult for an intruder to retrieve the original content of each block without a key, but an intruder can still infer partially correct order of the blocks within each shred with high accuracy. In security, the less information to the outside world is better, and so we would like to maintain a random order of blocks among different shreds which essentially should not follow a regular predictable pattern, and thus we aim at replacing the round-robin dispatching of blocks among threads.

One way to achieve that is to design and implement a lottery scheduler. Initially, the target lottery scheduler should maintain a list of all blocks. Each thread will invoke the lottery scheduler prior to invoking the File Spooler. The lottery scheduler will generate a ticket for the thread with a random number. At any point in time, each thread should have a ticket, and the thread with the lowest ticket number will be given a block number by the scheduler to process, which the scheduler chooses at random and mark this block as used so it is not given to any other thread in the future; **obviously this operation needs to be atomic**. Essentially, a thread with a block number ready to serve should first ask for the next ticket before invoking the FileSpooler requesting the block. This will ensure that each thread will have a ticket at any point in time.

Obviously, you need to modify the original FileSpooler, you already implemented in phase 1, to have a block number as a parameter to serve. Each time a thread processes a block, it will add its information in the corresponding element in the Index Queue. It is obvious that in this case the shreds might not be equal in size with respect to the number of blocks stored in each. The diagram below gives a high level overview of the import process.



It is very important to highlight that upon finishing importing a file into the Safe Box it is essential to store the Index Queue into a separate file with the same name of the imported file but with ".index" extension. Obviously, it is quite important to store the content of the queue in an encrypted format using the same key. A revised steps carried out by each thread of the import scenario can be summarized as follows:

1. Upon starting each thread will request a Ticket from the Scheduler.
2. Wait for thread to be selected by the scheduler based in its Ticket.
3. Request another ticket.
4. Request Block from Scheduler.
5. Read Block from file.
6. Encrypt Block.
7. Store Block in corresponding shred.
8. Update Index Queue.
9. Check scheduler for more blocks
    1. Go to step #2 if more blocks are available.
    2. Exit if not.

IMPORTANT: The above steps are described from the functional point of view and will need to be implemented with multithreading constructs in mind which might entail dome modifications and deviation from the above order.

In the case of the export, you will need first to load the Index Queue from the file it is stored into and decrypt it. You will then start a number of threads equivalent to the number of shreds. Threads will start dequeuing items from the queue, one at a time, which needs to be essentially synchronized. Based on the information stored in the queue item the thread will read the lock from the corresponding shred at the correct location, decrypt the block, and store its decrypted version into the correct location in the exported file. It is very important to highlight that in the case of export, a thread does not operate on a single shred file, rather it will alternate based on the queue item it processes

at any point in time, and hence high measures of synchronization need to be applied.

Finally, you need to handle all errors and report them to the user. You are required to use exception handling to achieve that and to build an exception handling hierarchy to be able to extend the handling of exceptions and error handling as functionalities are added.

What to submit

1. All your code.
2. Your code should be split among header files (.h) and source files (.cpp), and all your implementation need to be in the source files. You need to have a very good reason for including implementation in header files, and an explanation of such motives need to be included in your report.
3. Very detailed in-code documentation.
4. A modular Makefile hierarchy.
5. A report describing how to build the software using g++, including all design decisions taken and their alternatives, and explaining anything unusual to the teaching assistant. The report should be in PDF format.
6. Contributions files; each group member will provide a file named after her/his name explaining what she/he did and her/his contribution in this project phase.
7. Do not submit any object, or executable files. Any file uploaded and can be reproduced will result in grade deductions.

How to submit:

First of all, you will need to commit your code to your GIT repository. You will need to upload a text file named phase2_git_commit_id.txt, that includes your GIT commit ID to blackboard. The commit ID submitted by your group on blackboard will be used by the TA to grab your code which will be considered and graded, so make sure to include the correct version commit id. The TA will compare the commit ID date on the GIT repository with the deadline to know if the code was committed on the GIT repository before or after the deadline.

Moreover, you are required also to compress all your work: source code, report, readme file, and any extra information into a zip archive. You should name your archive in the specific format <Section_ID>_<Team_ID>_Term_Project_F19_Phase2.zip. Finally, upload your archive to blackboard.

If you were able to package your work into a docker environment, which is optional, you can mention your docker repository in your readme file, but it is a must to upload all your source code assignment material to blackboard and commit you code as well to your GIT repository.

This is a group project and you should elect a group captain among your group members, and all the submissions are expected to be performed through her/his blackboard account. IMPORTANT: EVERY GROUP SHOULD SUBMIT THROUGH THE GROUP CAPTAIN ONLY.

IMPORTANT: You will have to present your work and run a demo to the TA and/or the Professor in order to get your grade.

Grading

This project phase is worth 25% of the overall project grade. This phase will be graded on a 100% grade scale, and then will be scaled down to the 25% its worth. The grading of this phase will be broken down according to the following criteria:

a. Safe Box Program		80%
i. Functionally running	50%	
ii. Code Quality	15%	
iii. In-Line Documentation	15%	
b. Report		20%

The work will be evaluated based on the correct execution of the safe box program based on the functionalities stated in this phase, the code correctness and neatness, design decision aspects, and how much you have utilized the concepts of abstraction, encapsulation, modularity, and hierarchy. It is very important to highlight that although in-code documentation is worth 10%, yet missing or unclear documentation might result in more grade deductions if the grader cannot verify the correctness of your code and logic, and your provided in-line documentation were not sufficient to clarify such matters.

Delays

You have up to 3 calendar days of delay after which the corresponding phase will not be accepted and your grade in that case will be ZERO. For every day (of the 3 allowed days per phase), a penalty of 10% will be deducted from the total grade.