Highly Dependable Systems

Sistemas de Elevada Confiabilidade

Project 1: File server with integrity guarantees

Goals

The goal of the project is to implement a file system providing various different dependability guarantees, which will be strengthened throughout the course of stages 1 through 3.

The file system has a simplified specification, namely since it supports a one to one mapping between files and writers: each file can be written by a single entity (client), and each entity can write to a single file. There are no restrictions concerning who can read a file. Each client has to write sequentially, i.e., you can assume that each client does not issue several writes in parallel.

The file system interface is also simplified, since it consists of functions to initialize a client, to write to the client's file (receiving as arguments the position, the length in bytes, and a buffer with the new contents), and to read a file (receiving as arguments the identifier of the file, the position where to read and the number of bytes to read). The value that is output by a read should reflect the effects of the sequence of previous writes that completed before the read started (and it may or may not reflect writes that happen concurrently).

In this file system, the identifier of the file is a cryptographic hash of the public key of the client that is allowed to write to the file. This will help in the process of checking the integrity, as explained next. The full interface of the file system is as follows:

- FS_init → returns id
  Specification: initializes the file system before first use. In particular, it should make the necessary initializations to enable the first use of the cryptographic primitives. Returns the identifier that will be associated with the file of this client.
- FS_write(int pos, int size, buffer_t contents)
  Specification: writes the data contained in "contents", of size "size", at the position "pos" of the file associated with the client that invokes it. If the file size was smaller than pos+size then it must be increased. If the file size was smaller than pos then it must be padded with zeroes.
- FS_read(id_t id, int pos, int size, buffer_t contents) → returns bytes_read
  Specification: attempts to read "size" Bytes of the file with identifier "id" (which you can assume is obtained through an out of band mechanism), at position "pos", and places the returned data in the buffer "contents", returning the number of bytes that are actually read (which may be smaller than "size" in case it tries to read beyond the end of file).

Design requirements

The design of your file system is split into two parts: a library that is linked with the client, and a block server that is responsible for storing blocks with file contents. The library is a client of the block server, and its main responsibility is to translate requests to read and write files into requests to store (put) and retrieve (get) blocks on the server. The block server, in turn, exports a simple interface with three different functions: get, put_k, and put_h. The specification of these functions is as follows:

- get(id_t id) → returns data

  specification: returns a data block that was previously stored with the identifier id. The implementation should guarantee that integrity of the returned data is preserved, i.e., the contents of the data block are the same that were given as input to the "put" operation. An error must be returned under unexpected conditions, namely if the server did not return a block with integrity guarantees.

- put_k(data_t data, sig_t signature, pk_t public_key) → returns id

  specification: Stores a public key block with the contents contained in "data", signed with a signature contained in "signature", which can be validated using "public_key". An error must be returned under unexpected conditions, namely if the signature is not valid. The id associated with the data block that is returned is a hash of the public key passed as a parameter.

- put_h(data_t data) → returns id

  specification: Stores a content hash block with the contents contained in "data". The id associated with the newly created data block that is returned is a hash of that block.

As an initial design (for partial credit only), you can store the entire file on a single public key block using the put_k function. However, for full credit, you need to meet an additional performance requirement that the file system should be able to support very large files (e.g., on the order of GBs) without requiring you to sign the entire contents of the file. For that, you need to split the contents of the file into public key and content hash blocks. There is no single right way to do this, so it is up to you to propose a design and justify why it is adequate.

Implementation requirements

Your project has to be implemented in Java using the Java Crypto API for the cryptographic functions.

We do not prescribe any type of interface between the client and the block server. In particular, you are free to choose between using sockets, a remote object interface, or a SOAP-based web service.

Implementation Steps

To help in your design and implementation task, we suggest that you break up this task into a series of steps, and thoroughly test each step before moving to the next one. Here is a suggested sequence of steps that you can follow:

Step 1: Block server with no integrity guarantees – Design, implement, and test a block server (and a trivial test client) with the interface above, but that does use the crypto parameters, i.e., does not sign the contents, validate signatures, or compute hashes of the data (the returned id can be a random number, for instance).

Step 2: Complete block server – Add the missing functions to the block server, namely id and signature generation and the corresponding validation.

Step 3: File system with a single block – The next step is to implement the file system functionality, but using only a single public key block per file.

Step 4: Improve performance – Finally, you should optimize the design from step 3 by breaking up each file into both public key and content hash blocks, as mentioned above.

Submission

Submission will be done through fenix. The submission shall include:

- a self-contained zip archive containing the source code of the project and any additional libraries required for its compilation and execution. The archive shall also include a set of demo applications that demonstrate the mechanisms integrated in the project to tackle security and dependability threats (e.g., detection of attempts to tamper with the content of data).
- a concise reports of up to 4,000 characters addressing:
    - explanation and justification of the FS design
    - explanation of the integrity guarantees provided by the system
    - explanation of other types of dependability guarantees provided

The deadline is March 11, at 17:00. More instructions on the submission will be posted in the course page.