Advanced Topics in Online Privacy and Cybersecurity

ORAM

Amit Roth

API

The project code provides 2 main files you need to deal with.
server.py, and the function Server.run().
client.py, and the functions Client.read(name) and Client.write(name, block)

The communication between the client and there server executed by python sockets, and provides security features:

1. End to end encryption, using cryptography.fernet
2. ORAM protocol, oblivious RAM to prevent statistical inference.
   Implementation based on the article Path ORAM.
3. Hash authentication - to validate the reliability of the data.

DESIGN

As mentioned above, we have 2 files that the user interacts with, server.py and client.py. besides those 2 files we have clientAgent.py, block.py, bucket.py, settings.py and benchmarks.py. We will describe each file.

block.py - the smallest data structure that the server works with. Each block is represented by a name, and contains data as bytes. Note that the size of the data is not limited, but you should treat it carefully to prevent creating too large buckets that the socket cannot handle.

bucket.py - Bucket serves as a list of blocks. You can write blocks to the next available index using write_block(block). Note that in contrast to Block, the size of bucket is fixed, and you can change it in settings.py.

server.py - Server is responsible for communication and storing the data. Contains a list of N buckets, can read and write them, and communicate

using python sockets with a simple protocol for transferring read/write/close commands.

clientAgent.py - clientAgent is the highlight of this project. clientAgent is responsible for the communication, encryption, ORAM, and authentication. Specify the size of the server and start using clientAgent. With establish_connection() you connects to there server's socket, run initialize_structures and then you can start writing and reading. We won't explain every function on clientAgent because most are self explaining. Our 2 main functions which basically the same are oram_read(name) and oram_write(name, block). Each call, we take the path from a leaf to the root which contains the cell of our block (if previously been written), decrypts the blocks, add our block to a bucket or return the specified block, encrypts again and push to the same path.

Client.py - Client designed to minimize the api and the complexity of clientAgent. Client has 2 functions, write(name, block) and read(name). Just initialize Client(N) and start using the functions.

settings.py - in settings.py you can find settings of the ORAM protocol, communication and logs. Intended to prevent from changing constants in 2 places when running server and client from same computer.

benchmarks.py - the code used to calculate the benchmarks that you will see bellow. You can also look at the good as an example of using Client.

BENCHMARKS

Benchmarks result created by benchmarks.py.

Throughput [req/s]	Latency [s]	Size N	Number of requests
452.3	0.0017	31	10
344.5	0.0021	31	20
335.8	0.0031	31	30
493.5	0.0017	31	40
419.3	0.0019	31	100
345.5	0.0032	63	50
342	0.0024	63	100
358.6	0.0025	63	150
363.3	0.0023	63	200
379	0.0023	63	250
294.5	0.0027	127	100
291.8	0.0029	127	200

MULTICORE

Multicore can improve the run of the algorithm if implemented carefully. We cannot preform several requests at same time, because some buckets in the tree can be overridden. What you can do is to concurrence the write and read of specific blocks to the tree i.e. if we want to read the entire path [0, 1, 4, 9, 20], we can read each index at parallel using different threads, and to save the time that wastes on waiting to recv(). Note that this implementation is not that trivial, if you want to send requests at parallel you should open a different socket for each thread, otherwise 2 threads can speak at same time and destroy the protocol.