Secure Cloud Computing (202000026)
Q1 2025/2026
Issue Date: 29 September, 2025; **Due Date: 12 October 2025, 23:59 CEST**

Assignment 2: ORAM and Binary Fuse Filters

Introduction

In this assignment, you will study how clients can access remotely in an oblivious way for storage-as-a-service scenarios. It is recommend to solve both tasks of this assignment with an implementation in Python; however, if you prefer another programming language then feel free to use this alternative with a brief motivation why you decided to do so. ;-)

Submission Guidelines

This assignment will be done in pairs (groups of size 2). You have to hand in a document(PDF) with your answers and your source code with the program you implemented (plain text) on Canvas. Please mention your own name and student number as well as the name and student number of your partner. Do not submit your solution using multiple accounts. You may also submit an archive (ZIP/GZip) containing the report and source files, but please do not submit a RAR file.

Important Do not submit PDFs of hand-written documents (such as scans or photos) as we will not take them into consideration. Instead write your documents using LaTeX,MS-Word or similar, and then save/export as PDF.

1 Path ORAM *(14 Points)*

The goal of ORAM is to hide the data access pattern (which blocks were read/written) from the cloud storage server. From the server's perspective, the data access patterns of two sequences of read/write operations with the same length must be indistinguishable. Stefanov et al. [1] have proposed Path ORAM, a simple ORAM protocol where each data access can be expressed as simply fetching and storing a single path in a tree stored remotely on the server.

As discussed in the lecture, data is divided into N blocks and encrypted before it is outsourced to a cloud server. Each block contains B bits of data and is identified with a unique ID a in range $[0, N)$. Data is accessed in block units by either read or write operations.

On the server-side, data is stored in a binary tree, where each node is a bucket that can contain up to Z real blocks. Let $L = \lceil \log_2 N \rceil$ be the height of the tree and levels are numbered in $[0, L]$, where the root is on level 0 and the leaves are on level L. Let $x \in \{0, 1, \ldots, 2^L - 1\}$ denote the x-th leaf node in this tree, then any such leaf defines a unique path from x to the root. We use $\mathcal{P}(x)$ to denote the set of buckets along this path. Additionally, $\mathcal{P}(x, l)$ denotes the bucket in path $\mathcal{P}$ at level l of the tree.

The client stores two data structures:

Stash: During a read/write access, the stash stores blocks in $\mathcal{P}(x)$ retrieved by the client before it is evicted again. After every access, the stash may still hold some blocks that are overflowed from the tree buckets on the server. This data structure is usually empty or contains only a small number of nodes after a ORAM access is completed.

Position map: This is an array that associates each block a to a leaf node. For example $\texttt{position}[a] = x$ means that block a resides in some bucket in path $\mathcal{P}(x)$ or in the stash. The position map is updated over time as blocks are accessed and remapped.

1.1 Simulate Path ORAM

Your task is to simulate a Path ORAM. There is no need to implement a remote server or encrypt any real data blocks, but the goal is to evaluate the stash behavior for different parameters.
The client's position map is filled with independently chosen random numbers between 0 and $2^L - 1$; please use Python's `random` module to sample those numbers randomly.
Read and write simulation of a block should follow the algorithm given in pseudocode in Algorithms 1. Specifically, the client reads block a by performing $\texttt{data} \leftarrow \texttt{Access}(\texttt{read}, a, \texttt{Null})$.
For writing $\texttt{data}*$ to block a, the client executes the algorithm $\texttt{Access}(\texttt{write}, a, \texttt{data}*)$.
The Path ORAM access algorithm consists of the following four steps:

- Remap block: Store the old position of block a in x before remapping this block to a new random position.
- Read path: Read the path $\mathcal{P}(x)$ containing block a (if a is not stored along the path, then it must be in the stash).
- Update block: Update the data stored for block a if the access is a write operation.
- Write path: Write the blocks from the path back to the tree and possibly include some additional blocks from the stash if they can be placed into the path. (In practice, blocks should be re-encrypted using fresh randomness so they appear new to the server). Buckets are greedily filled with blocks in the stash in the order of leaf to root, ensuring that blocks get pushed as deep down into the tree as possible.

In the ORAM with read path eviction, blocks are evicted from the stash on the same path that was read to retrieve the block.
When performing eviction, it is important that blocks are placed in a position that is consistent with the position map (that is only updated when a block is accessed). Therefore, block a' from the stash can be placed in the bucket at level l on a specific path $\mathcal{P}(x)$ only if the path $\mathcal{P}(position[a'])$ to the leaf of block a' intersects path accessed $\mathcal{P}(x)$ at the considered level l. In other words, if $\mathcal{P}(x, l) = \mathcal{P}(position[a'], l)$ as stated in line 11 of Algorithm 1. If more than Z real blocks are to be stored in this bucket, they are left on the client's stash (and might be placed in other buckets in the following iterations).

1.2 Stash Size Analysis

The original Path ORAM description does not state how big the client stash has to be in order for the algorithm to never stop. Indeed, it is possible that the client might have to even store almost all the data stored in the ORAM in the local stash (as worst case

Algorithm 1 `Access`(`op`, a, data*)

1: $x \leftarrow$ `position`$[a]$ ▷ Remap block a
2: `position`$[a] \leftarrow$ UniformDistribution$(0, \ldots, 2^L - 1)$

3: **for** $l \in \{0, 1, \ldots, L\}$ **do** ▷ Read path for leaf x
4: $S \leftarrow S \cup$ ReadBucket$(\mathcal{P}(x, l))$
5: **end for**

6: $data \leftarrow$ Read block a from$S$ ▷ Update Block
7: **if** `op` $= write$ **then**
8: $S \leftarrow (S - \{(a, data)\}) \cup \{(a, data*)\}$
9: **end if**

10: **for** $l \in \{L, L-1, \ldots, 0\}$ **do** ▷ Write Path
11: $S' \leftarrow \{(a_0, data_0) \in S : \mathcal{P}(x, l) = \mathcal{P}($`position`$[a_0], l)\}$
12: $S' \leftarrow$ Select $\min(|S'|, Z)$ blocks from S'
13: $S \leftarrow S - S'$
14: WriteBucket$(\mathcal{P}(x,), S')$
15: **end for**

16: **return** `data`

consider what would happen if the client is so unlucky that UniformDistribution in line 2 of the algorithm always returned the same leaf). But since the algorithm is randomized, the stash size required in practice will be much smaller than the worst case. You are asked to analyze the appropriate size that needs to be allocated on the client for the ORAM using your simulation.

1.3 Data Collection for Stash Size

You are asked to implement an ORAM simulation (there is no need to encrypt or store data remotely) on at least **five million** access operations. Random access operations – both read and write (the operation should also be sampled randomly) – happen for all the blocks of memory in sequence: $\{1, 2, 3, \ldots, N, 1, 2, 3, \ldots, N, 1, 2, 3, \ldots, N\}$. This sequence was chosen because it is a worst case scenario in terms of the stash size. After the first 3 million accesses (which are used to "warm up your ORAM" such that it enters steady distribution), please start recording the stash size after each access.

At the end of your simulation, please write the collected data to a text file. The first line of the file should contain the line $-1, s$, where s is the total number of accesses that you run the simulation for (excluding the first 3 millions warm up accesses as explained above). From the second line, each line should contain i, s_i, where i is an increasing integer representing the stash size, and s_i is the number of accesses after which the stash had size greater than i (starting with $i = 0$ and up until when the maximum size of the stash you encountered is reached).

For example, if you run your ORAM for $3'000'003$ accesses and got stash size 1 after the $3'000'001$-st access, stash size 2 after the $3'000'002$-nd access, and again stash size 1 after the $3'000'003$-rd access, then you had 3 accesses with stash size greater than 0, one access with stash size greater than 1 and no accesses with stash size greater than 2. Thus, your

file should look like:

−1,3
0,3
1,1
2,0

Tasks

1. Implement an ORAM simulation so that you can collect data as described in Section 1.3.

   Please collect data for the following two configurations:

   (a) $N = 2^{16}$, bucket size $Z = 2$

   (b) $N = 2^{16}$, bucket size $Z = 4$

   Pick the remaining parameters accordingly, i.e. $L = 16$. Please submit the files together with your source code and name your text files `simulationX.txt`, where $X \in \{1, 2\}$. *(10 Points)*

2. Using the collected data, create a graph for both bucket sizes $Z = 2$ and $Z = 4$, where the x-axis expresses the required stash size R (excluding stash size 0) and y-axis expresses the probability the stash size is larger $Pr[\mathsf{size}(S) > R]$. *(4 Points)*

2 ChalamentPIR *(6 Points)*

Private Information Retrieval (PIR) allows a client to query one value of a database without leaking the index they query for. PIR can be implemented with homomorphic encryption (which will be introduced in one of the following lectures). Specifically with homomorphic encryption, the client can encode the query as one-hot encoding: that is, a vector $q = (0, \ldots, 0, 1, 0, \ldots, 0)$ of length n with exactly one 1 at position i and the remaining values are all zero. Then all values are encrypted as $c_i = \mathsf{Enc}(pk, q_i)$ and sent to the server. The server can multiply the database values $v = (v_0, \ldots, v_n)$ with the encrypted queries, and does not learn the set bit at position i; due to the homomorphic property it holds that $c_i \odot v_i = \mathsf{Enc}(pk, q_i \cdot v_i)$. All such calculated values are aggregated to $\sum_i^n c_i \odot v_i$ and returned as final response which can be decrypted by the client. Celi and Davidson [2] use Binary Fuse Filters (BFF)s to compress the query to be sublinear and supports keyword queries (instead of index queries). Originally, filters such as Binary Fuse Filters and the more popular Bloom Filters (BFs) have been introduced for probabilistic membership tests, that is, given a value x we can check efficiently if x is likely in S for a large set S.

2.1 Bloom Filters

A standard Bloom Filter consists of k hash functions $h_1, \ldots\ h_k$, with $h_i : \{0,1\}^* \mapsto [N]$ (that is, any binary string is mapped to a number $n \in \{0, \cdots N-1\}$ for all i. Further, it

consists of a bit-array F with length N initialized with all zeroes. To insert an element s into the Bloom Filter we set the corresponding bits:

$$\begin{aligned} F[h_1(s)] &= 1 \\ F[h_2(s)] &= 1 \\ &\vdots \\ F[h_k(s)] &= 1 \end{aligned}$$

which is repeated for all $s \in S$. For a set H of hash functions $h_1, \ldots, h_k$ we write $H(x)$ to denote the evaluation of $h_1(x), h_2(x), \ldots, h_k(x)$; similarly, we can then write $F[H(x)] = 1$ to abbreviate the procedure to insert element x.
In order to check if value x is likely in S we check if the corresponding bits are all set:

$$(F[h_1(x)] == 1) \text{ and } (F[h_2(x)] == 1) \text{ and } \ldots \text{ and } (F[h_k(x)] == 1).$$

Again, we can abbreviate this check as $F[H(x)]$. If we use k hash functions, we need to check k random bits within F.
We can implement hash function h_i using a Hash-Based Message Authentication Code (HMAC) that uses a hash function $G : \{0,1\}^* \to \{0,1\}^\lambda$ and key κ, denoted as $\mathsf{HMAC}_G(\kappa, \cdot)$, as follows: Given N that is a power of two, fix key κ_i and interpret the output of G as integer in $\{0, 1, \cdots, 2^\lambda - 1\}$. Then calculate

$$h_i(x) = \mathsf{HMAC}_G(\kappa_i, x) \pmod{N}.$$

2.2 Binary Fuse Filters

In comparison to Bloom Filters, a Binary Fuse Filter of length N is distributed in segments, one can imagine these segments as a collection of multiple Bloom Filters $B_0, B_1, \ldots$ each with bit array of length s. Value x is inserted to the BFF, by selecting a fixed segment using function $\sigma(x)$ and then random bits in this Bloom Filter $B_{\sigma(x)}[H(x)]$ are set as described previously in Section 2.1. Celi and Davidson [2] propose a modified version of BFFs; a simplified version of the setup algorithm is described in Algorithm 2, refer to the original paper for full details. The (imagined) Bloom Filter selection and filling is described in Algorithm 2 in lines 8 – 11 [1].
Their modified version does not only support membership checks, but also storing values in the BFF. Specifically, they consider values $v \in \mathbb{Z}_p$, and then insert key,value pairs (q, v) in such a way that:

$$v = \sum F[H(q)] \pmod{p}$$

2.3 String Representation

In this assignment we will use strings as keys, however, hash functions have byte arrays as input and output. We encode all strings in UTF-8 (lower and upper case matter here) and if we interpret a byte array as integer, we do so in big endian, that is, the most significant byte is at the beginning of the byte array. For example, the string `ABC` consists of three bytes $0x41, 0x42, 0x43$ in hex representation. This byte array is then represented

[1] Note that we start counting at 1 here.

Algorithm 2 `BFF_Setup`

Require: A parameter $k \in \{3, 4\}$. A parameter $m \in \mathbb{N}$ denoting the number of keys in the maps written to BFF_p.

Set $s \in \{2^{\lfloor \log_{3.33}(m)+2.25 \rfloor}, 2^{\lfloor \log_{2.91}(m)-0.5 \rfloor}\}$ for $k \in \{3, 4\}$ respectively.
Set $N = 2m$
Sample universal hash functions $h' : \{0,1\}^* \mapsto [N/s]$ and $h'' : \{0,1\}^* \mapsto [s]$
$F = \emptyset$ ▷ This is an empty list
for $i \in [N]$ **do** $F[i] \xleftarrow{\$} \mathbb{Z}_p$
end for
$H = \emptyset$
for $i \in [k]$ **do**
    Let h_i be the function that is evaluated as $h_i(\cdot) = (N/s \cdot (h''(\cdot) - 1)) + h'(\cdot||i)$.
    $H[i] = h_i$.
end for
return (F, H).

by integer $65 \cdot 256^2 + 66 \cdot 256^1 + 67 \cdot 256^0 = 4276803$. Hence, in order to represent a string of length ℓ, we set $p = 2^{8 \cdot \ell}$ in Algorithm 2. If we concatenate a string with an integer, then this integer is first parsed to a string. For example, $A||1$ corresponds to integer 16689.

Tasks

1. What is the HMAC of the (value-)key `assignment1` with (HMAC-)key `M4Ck3y` using SHA256? *(1 Points)*

2. We want to store strings in a Bloom Filter B of length $N = 2^{10}$. As hash function we define $h_i(x) = \mathsf{HMAC}_{\text{SHA256}}(\texttt{BloomyKey}, x||i)$ and we use $k = 4$ hash functions. What are the 4 bits that must be set for $x =$ `Bloom filters are great`, i.e. the bits $B[H(x)]$? *(2 Points)*

3. A BFF F that has been setup as described in Algorithm 2 has been created and can be downloaded as file `bff.txt` shared on Canvas. Integers are separated by comma, and the i-th integer represents $F[i]$. The parameters of BFF as set as follows:

   - $p = 2^{64}$
   - $k = 3$
   - $m = 262144$
   - $h'(\cdot) = \mathsf{HMAC}_{\text{SHA256}}(\texttt{confidential}, \cdot) \pmod N)/s$
   - $h''(\cdot) = \mathsf{HMAC}_{\text{SHA256}}(\texttt{s3c0nd\_s3cr3t}, \cdot) \pmod s$

   Reconstruct the string that has been encoded in the BFF under the key `assignment`. *(3 Points)*

References

[1] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious

RAM Protocol. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 299–310, 2013.

[2] Sofía Celi and Alex Davidson. Call me by my name: Simple, practical private information retrieval for keyword queries. In Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, pages 4107–4121, 2024.