Advanced Topics in Online Privacy and Cybersecurity - 67515

Tree based – PATH ORAM implementation

Reut Ora Oelbaum

The implementation of the algorithm is based on the lecture of the lesson

[Lecture 2: Oblivious RAM - Google Slides]

The demands for the project:

Programming task (functionality)

- Implement client and server applications
- Server initialized to support N datablocks
- Client can store data associated with a name on server
- Given a name, client can retrieve associated data (null if doesn't exis
- Client can delete data associated with name from server
- Client has small memory (can't store data)
- **Use existing libraries for: Hash functions (HMAC in particular), Enc/Dec algorithms.**

Programming task (security)

- Server is oblivious to data content and client's access patterns
  - End to end encryption, ORAM
- Server can't trick client into accepting corrupt or outdated data
  - Authentication

Programming task

- Submit code, design document
- Include performance benchmarks
  - Throughput (number of requests/sec) vs. N (DB size)
  - Latency (time to complete a request) vs. throughput
  - Does your implementation benefit from multicore?

API

מצורפים הקבצים הבאים:

client_TCP.py*

server_TCP.py*

כנדרש האלגוריתם ממומש עבור 2 משתמשי קצה- שרת ולקוח. (כרגע בקוד יש תמיכה ללקוח בודד אך ניתן להרחיב לתמיכה במס' רב של משתמשים).

ע"מ להריץ את הפרוייקט- יש להריץ קודם את הקוד של השרת ורק לאחר מכן את של הלקוח.

Libraries

בקוד נעשה שימוש בספריות היחסיות בסיסיות הבאות:

```
math, random, string
```

וכמו כן נעשה שימוש בספריות הבאות:

- Socket
  מאפשרת ליצור תקשורת בין 2 תהליכים, אפליקציות שונות- בפרט במקרה של לקוח שרת כמו בפרוייקט זה.

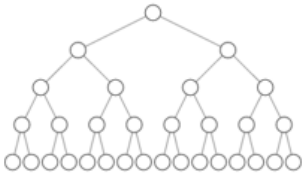
- Pickle
  ספרייה שאחראית להפוך אובייקט מורכב לרצף בייטים ולהפך. לדוג' בפרוייקט זה יש אובייקט שמייצג צומת בעץ- הלקוח הופך אותו לרצף של בייטים בעזרת הספרייה, מצפין רצף זה ולאחר מכן שולח רצף זה לשרת.

- Hashlib
  ספרייה שעושה HASH על מידע באופן שמאפשר אימות שלמות הנתונים. הלקוח שומר HASH של תוכן הקובץ ובכל פעם שהוא קורא/כותב לקובץ זה הוא מוודא שהשרת לא שינה את התוכן (הוא מקבל את התוכן מוצפן- מפענח אותו ומחשב מחדש את ערך הHASH ומשווה לערך השמור אצלו)

- Cryptography.fernet
  משמשת להצפנה, לפיענוח וליצירת מפתח סימטרי שמשמש להצפנה ולפיענוח.

Design & Architecture

שרת



- עבור כל צומת נשמרת מחרוזת שנראית חסרת משמעות (מידע שמוצפן)
- העץ ממומש בתור מערך לצורך גישה לצומת ובפרט עלה ב o(1)

לקוח

נשמר מיפוי בין שם של קובץ לעלה שאליו הוא משויך ולHASH על תוכן הקובץ (מס פרמטרים נוספים- כמו לדוג' אורך הCONTENT המקורי והאינדקסים של הצמתים על המסלול)

שרת:

```
while True:
    data = client_socket.recv(MESSAGE_CONNECTION_SIZE)
    if not data:
        break
    data = data.decode('utf-8')
    print(data)
    if data == "send root":...
    if data == "get node":...
    if data == "post node":...
    if data == "print tree":...

client_socket.close()
```

כפי שניתן לראות בתמונה לעיל השרת תומך ב4 פעולות כל עוד החיבור בינו לבין הלקוח פעיל-

- Get Node – ופעולה נוספת שהיא מקרה פרטי- Send Root (הוגדר כפעולה נפרדת כי היא נפוצה ומטעמי נוחות)
- Post Node

כאשר מבחינת רעיון ממומש בדומה לפרוטוקול HTTP- בGET מקבלים NODE ספציפי ובPOST שולחים NODE חדש במקום NODE שכבר קיים בעץ.

פעולה נוספת שהשרת תומך היא הדפסת העץ שלו- נח בעיקר לצרכי דיבוג.

לקוח:

נתאר בקצרה את הפונקציות/ פעולות שהשרת תומך בהן:

בפעולות על האינדקסים ניתן למצוא את הפונק' הבאות: (שניתן להבין מהשם מה תפקידן.)

```
def get_leaf_index(branch):

def get_parent_index(node):

def get_left_child_index(node):

def get_right_child_index(node):

def get_path_indexes(leaf_id):
```

פוקנציות נוספות הן הפונקציות שאחראיות על התקשורת וכוללות את:

```
# communication
def get_root_node():

def get_node(index):

def get_path(leaf):

def print_tree():
```

ניתן לראות את ההתאמה בין ה4 פונק' הנ"ל לבין התמיכה של השרת ב4 פעולות.

```
def padding(content):
    return content.ljust(len_of_file, 'x')


def unpadding(content, len):
    return content[:len]
```

בנוסף השרת משתמש בפונק' ריפוד וההופכית לה- הסיבה לכך- להסתיר מהשרת גם את הגודל של הקובץ. נעשה שימוש בפונק' הנ"ל גם על שם הקובץ וגם על תוכן הקובץ וככה יוצא שגודל המחרוזת שנשמר בשרת בכל צומת הוא **קבוע**.

כמו כן אצל הלקוח יש פונק' שעוטפות את הקריאה לפונק' תקשורת השונות (שמוסיפות הצפנה, אימות ומוסיפות בדיקות תקינות- לדוג' שמשכתבים/ מוחקים קובץ שקיים ושלא עוברים את מכסת הקבצים שהמערכת תומכת בה).

הרצה בצד הלקוח:

מתאפשרת ב2 צורות שונות-

1. באופן ידני לכתוב ב main את הפעולות שנרצה לבצע (ובפרט לפעולת הכתיבה ניתן לתת ערך מראש- מבלי לחכות לקלט מהמשתמש)
2. בעזרת ממשק משתמש פשוט-
מציג למשתמש את הפעולות האפשריות בכל שלב ומידע נוסף (לדוג' מהם שמות הקבצים הקיימים במערכת)
להלן דוג' הרצה: (הוספת קובץ, קריאה שלו, שינוי תוכן וקריאה מחודשת - בין פעולה לפעולה מודפסות שוב הפעולות הנתמכות- ערכתי ופה זה לא מופיע במטרה לשמור על תמציתיות)

```
choose num of suppurted action:
1. add file
2. read file
3. re- write file
4. delete file
5. exit

1
Enter file namefile
Enter file contentreut
we add new file

2
Choose the file to read
The files in the system are:
1: file
file name:file
auth SUCCESS!
reut
```

```
3
Choose the file to re-write
The files in the system are:
1: file
file name:file
enter new content of file. len:100Oelbaum
auth SUCCESS!

2
Choose the file to read
The files in the system are:
1: file
file name:file
auth SUCCESS!
Oelbaum
```

Security

כנדרש:

- בתרגיל מימשנו הצפנה מקצה לקצה (end to end encryption)
- האלגוריתם Path Oram בנוי בצורה כזאת שהשרת לא חשוף לדפוסי השימוש של המשתמש.

(בקצרה- מתאפשר מכך שמחזירים בכל פעם מסלול (ולכן השרת לא יודע אם ניגשים לקבצים ישנים- ברמות נמוכות או לקבצים 'חדשים' (נוספו לאחרונה/ נקראו/נערכו לאחרונה), בכל קריאה/כתיבה לקובץ הוא עולה לשורש לאחר שהוא עובר הצפנה מחדש.)

- Authentication נעשה בעזרת sha224 - לאחר פיענוח המידע נפעיל מחדש את פונק' ה hash ונשווה לתוצאה ששמורה אצלנו.

Performance Benchmarks:

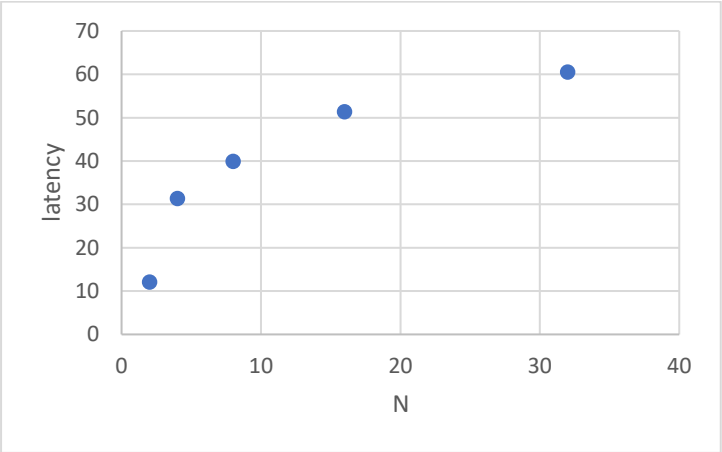
הסבר לגרפים הבאים-

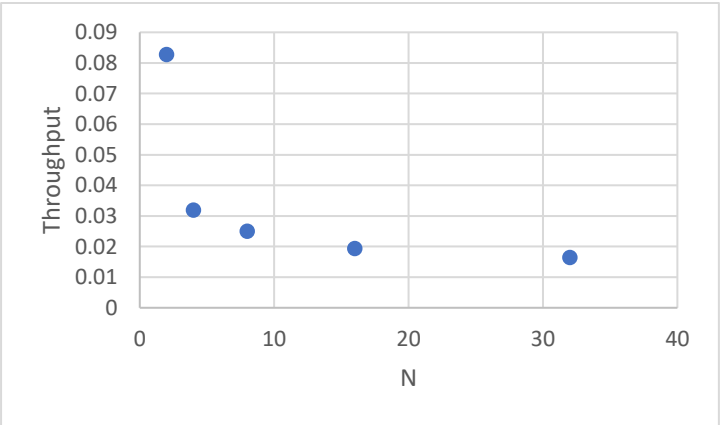
העמודות 5, 10, 15 נמדדו באופן מפורש- מדדנו כמה זמן לקחו 5/10/15 פעולות של קריאה (של אותו קובץ. נצפה לתוצאות דומות כאשר מדובר בפעולת כתיבה (כחלק מהעניין שמסתירים מהשרת גם האם התבצעה כתיבה/ קריאה). לעומת זאת פעולת מחיקה של קובץ קוראת רק בצד של הלקוח ולכן אינה דורשת כלל תקשורת והיא מהירה יותר.

העמודות לאחר מכן נעשו בצורת חישוב- כמה לקחו 30 הפעולות ביחד ואז נרמול לכמה לוקחת פעולה אחת (Latency). ומכך באופן ישיר נחשב את הthroughput

N	1	5	10	15	30
2	12.0804	60.3658	102.025	200.022	362.413
4	31.3338	110.7	253.84	575.476	940.015
8	39.9188	157.95	452.689	586.924	1197.56
16	51.4051	190.841	516.927	834.385	1542.15
32	60.5217	309.659	606.151	899.84	1815.65

Throughput (number of requests/sec)	Latency (time to complete a request)	N (Num of leafs in the tree-DB)
0.082778462	12.08043699	2
0.031914371	31.33384686	4
0.025050873	39.9187685	8
0.01945332	51.40510715	16
0.016523005	60.52167962	32





Multicore

בקוד שלי אין מקביליות מובנית. אבל ניתן להוסיף אותה- בזהירות.

לדוג'- כשקוראים קובץ- מקבלים את כל המסלול מהשורש לעלה שמאפיין אותו.

(במידה ולא עושים את הייעול של לעקוב באיזה רמה בעץ הקובץ נמצא- כמו שאני מימשתי)- ניתן לחפש את הקובץ במקביל בכל צומת במסלול כאשר הצמתים במסלול מחולקות על הליבות השונות.

(בייעול שלי- אמנם מקבלים את כל המסלול כדי להסתיר את דפוס השימוש מהשרת אבל מטפלים (מפענחים ומחפשים) רק בצומת ספציפית ששם הקובץ.

במידה והשרת תומך במס' לקוחות- ניתן למקבל באופן מוחלט- ניתן לחלק את הלקוחות על הליבות השונות ושהבקשות עבור הלקוחות השונים יקרו במקביל- כי השרת שומר ממילא עבור כל אחד עץ (DB) נפרד.