TECHNISCHE
UNIVERSITÄT
DARMSTADT

Master Thesis

Optimizing Private Information Retrieval for Compromised Credential Checking

Daniel Günther

November 22, 2019

Engineering Cryptographic Protocols
Department of Computer Science
Technische Universität Darmstadt

Supervisors: Prof. Dr.-Ing. Benny Pinkas
Prof. Dr.-Ing. Thomas Schneider

Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Daniel Günther, die vorliegende Master Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Thesis Statement pursuant to §23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Daniel Günther, have written the submitted Master Thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are identical in content.

Darmstadt, November 22, 2019

Daniel Günther

Abstract

Credential stuffing attacks allows an adversary to hijack an account published in a data breach. To prevent these attacks, the industry provides so-called Compromised Credential Checking (C3) tools that allow users to check if their credentials are leaked in a data breach. However, state of the art tools like HaveIBeenPwned (HIBP) and Google Password Checkup (GPC) Chrome extension (USENIX Security'19) leak a prefix of the hashed credentials of the user which is enough information to exploit a successful credential stuffing attack as shown by Li et al. (ACM CCS'19).

In this thesis, we give the first C3 protocol that achieves perfect anonymity, i.e., it leaks no information about the user's credentials. For this protocol, we use Private Information Retrieval (PIR) that allows a client to securely query a database entry while the server learns no information about the client's query. Since modern PIR schemes are not efficient enough for a real-world deployment, we introduce *Query-Dependent Preprocessing* PIR that moves $\frac{n-1}{n}$ of the online computation to an offline phase for $n \geq 2$ servers. We show that C3 with PIR is practical by implementing our query-dependent preprocessing optimizations. We measure the performance of the 2-server PIR part of our C3 protocol on a database with 2 billion entries, which results in 1.8 MB communication and 13 seconds runtime over a WAN network.

Acknowledgments

I would like to thank my supervisors Prof. Dr.-Ing. Thomas Schneider and Prof. Dr.-Ing. Benny Pinkas for their helpful discussions and comments in the last months. I am thankful for working with you together and I am looking forward for more joint works.

I also would like to thank my family and friends for supporting me all the time throughout my studies. Especially, I would like to offer my special thanks to Timo Feick for supporting and encouraging me to continue. Thanks for your understanding during so many times.

Contents

1 Introduction

Data breaches occur more and more in the recent years. These breaches contain highly sensitive information about users, e.g., their passwords. The most prominent breach contains more than two billions of password and is called Collection 1-5 [Gre19]. Thomas et al. [TLZ$^+$17] show that 6.9% of the breached credentials are still in use even on non-exposed platforms. This enables adversaries a basis for credential stuffing attacks, where the adversary compromises accounts by trying leaked passwords on other services. Usually, the affected platforms reset the passwords of their users after an expose, but this does not alert the users significantly about the risk of using the same credential on other platforms. So, there is a demand for tools that enable users to check if their credentials are breached - called *Compromised Credential Checking* (C3) tools [LPA$^+$19].

On the market, there exist two C3 tools called *HaveIBeenPwnd* (HIBP) [Shi19] and ENZOIC [ENZ16b]. Recently, Thomas at al [TPY$^+$19] published their *Google Password Checkup* (GPC) tool as Google Chrome extension that is the first C3 service protecting against malicious clients, who want to learn information about credentials from other people. They achieve this with the help of a Private Set Intersection (PSI) protocol that enables two parties to jointly compute the intersection of their secret sets without revealing anything but the result. However, all of these tools have the problem that they leak a prefix of the hashed credentials, since PSI is just used on a small subset of the whole database, namely all entries having the same prefix. A PSI protocol on the whole database would avoid such leakages, but it is too inefficient for large-scale databases.

Li et al. [LPA$^+$19] showed that the knowledge of the credential's prefix suffices to compromise up to 86% of the leaked accounts within 1000 attempts (even up to 73% of account that are not included in a data breach). They also provide two new C3 protocols that protect the user's sensitive information better but the transcript still leaks information that enables credential stuffing attacks.

Thomas et al. [TPY$^+$19] and Li et al. [LPA$^+$19] both suggest the use of so-called *Private Information Retrieval* (PIR) to provide a C3 tool that leaks nothing but the result. PIR allows a client to retrieve a database entry from a server, while the server learns no information about the query. However, [TPY$^+$19] and [LPA$^+$19] state that current techniques are not efficient enough to be operated in a real-world deployment. In this thesis, we demonstrate the opposite and show that we can use PIR in an efficient C3 tool that leads to perfect anonymity.

For our C3 protocol we need an efficient PIR scheme since this highly influences the performance. We have a closer look at RAID-PIR by Demmler et al. [DHS14; DHS17] that is a multi-server PIR scheme which extends Chor et al.'s PIR scheme [CGKS95], where $n \geq 2$ non-colluding servers share a database and each server interacts with the client. Those schemes are generally more efficient than single-server PIR schemes, but have the disadvantage that n servers are needed that are assumed to not collude. We operate RAID-PIR in a two server setting for our C3 protocol since two non-colluding servers is the easiest setting to realize to date. For this, we extend the existing RAID-PIR scheme into our new so-called *Query-Dependent Preprocessing PIR* model. This model lets the servers choose a part of the client's query and moves $\frac{n-1}{n}$ of the online computation to an offline preprocessing phase without loosing any privacy, where n is the number of PIR servers. We implement the RAID-PIR scheme with our query-dependent preprocessing improvements and obtain up to 40% better runtime than the original RAID-PIR scheme for $n = 2$ servers.

An application for PIR is in private set inclusion protocols, that check if an item is included in the database without letting the server learn the requested item. Example databases for these kind of application are medical data and patents. Demmler et al. also use PIR for their anonymous messaging service called *OnionPIR* [DHS17].

1.1 Contributions and Outline

Our main contributions are summarized below:

- We provide a new multi-server PIR model, called *Query-Dependent Preprocessing*, that allows per-query preprocessing by the servers and thus significantly reduces the online computation time (Sect. 3.3).
- We adapt the RAID-PIR scheme of Demmler et al. [DHS14; DHS17] in the query-dependent preprocessing model, implement it, and gain up to 40% better performance for $n = 2$ servers and up to 63% for $n = 3$ servers than the original scheme (Sect. 3.3.2) by moving $\frac{n-1}{n}$ of the online work to an offline preprocessing phase.
- We transfer the database compression idea of Tamrakar et al. [TLP+17] to PIR and call this *Compressible PIR* (Sect. 3.4).
- We adapt the C3 scheme by Thomas et al. [TPY+19] to gain the first privacy-friendly C3 protocol that leaks no information about the client's data while still protecting against malicious clients. For this, we use our query-dependent preprocessing RAID-PIR scheme with a compressed database (Sect. 4.3).
- We show the practicability of our C3 scheme with PIR by measuring the time-critical PIR part on a database with two billion passwords, which results in 1.8 MB communication and 13 seconds runtime for $n = 2$ servers (Sect. 6.3).

Outline. We give related work on PIR in Sect. 1.2.

In Chapt. 2, we provide our notations (Sect. 2.1) and background information, containing Private Set Intersection (PSI) in Sect. 2.2 and the method of four Russians Sect. 2.3.

Chapt. 3 contains details about the RAID-PIR scheme [DHS14; DHS17] as well as our optimizations. We firstly present our understanding of a PIR model in Sect. 3.1. Thereafter, we give a complete description of Demmler et al.'s RAID-PIR scheme [DHS14] with the improvement of [DHS17] in Sect. 3.2. Based on that, we introduce our new Query-Dependent Preprocessing PIR model and describe our improvement of the RAID-PIR scheme [DHS14; DHS17] in Sect. 3.3. In Sect. 3.4, we introduce Compressible PIR that can be applied to most PIR schemes. We finish this chapter with an analysis of the communication and computation complexity of the presented RAID-PIR variants in Sect. 3.5.

In Chapt. 4, we give related work on C3 and describe our new C3 protocol. Sect. 4.1 summarizes state of the art C3 tools and Sect. 4.2 describes the GPC protocol by Thomas et al. [TPY+19]. Our new C3 protocol based on PIR is given in Sect. 4.3.

Some details of our Query-Dependent Preprocessing RAID-PIR implementation are given in Chapt. 5, specifically, the construction of our framework in Sect. 5.1 and our realization of the most time-critical part - the huge number of XOR operations - in Sect. 5.2.

We evaluate our implementation in Chapt. 6. Therefore, we explain our setup in Sect. 6.1 for the PIR benchmarks in Sect. 6.2 as well as the benchmarks for the PIR part of our new C3 protocol in Sect. 6.3.

Finally, we conclude this thesis in Chapt. 7 and give some future work directions.

1.2 Related Work on Private Information Retrieval (PIR)

Several PIR constructions were developed in the last years. We distinguish between multi- and single-server PIR. Multi-server PIR requires multiple non-colluding servers and is often very efficient, while single-server PIR just requires one server but is less efficient. We briefly summarize the most relevant PIR schemes below.

Multi-Server PIR. The idea of information theoretically secure PIR and a first construction was developed by Chor et al. in [CGKS95]. Their construction relies on n non-colluding servers where each server receives a query from the client and sends a response to it. Goldberg provides another more robust PIR construction in [Gol07] that is based on Shamir's secret sharing scheme [Sha79] as well as the additively homomorphic encryption scheme of Paillier [Pai99]. A PIR scheme is called robust if the client receives the requested data even if some servers are corrupted. These two PIR schemes are compared in [OG11] alongside some single-server PIR schemes as well as the trivial solution, namely downloading the whole database. [OG11] conclude that the scheme of [CGKS95] has a better response time than [Gol07]. However, depending on the database size and the available bandwidth, the trivial solution of downloading the entire database is even more efficient than all tested PIR schemes. An implementation of [CGKS95]'s PIR scheme was provided by [Cap13]. It was demonstrated that their implementation can be used for downloading software updates from a server nearly as quickly as downloading it with FTP. Another PIR scheme called RAID-PIR that is based on Chor et al.'s construction [CGKS95] was developed and implemented by [DHS14; DHS17]. Their scheme is more efficient than Chor et al.'s scheme and allows multi-block queries. We detail the RAID-PIR construction in Sect. 3.2. Multi-block queries were introduced by Henry et al. in [HHG13]. They extend Goldberg's PIR scheme [Gol07] by replacing Shamir's secret sharing [Sha79] with the ramp secret sharing variant [BM84]. While the scheme is able to efficiently query for multi-block queries while keeping privacy, the robustness of the scheme is reduced. A verifiable PIR scheme which allows the client to detect cheating servers was proposed in [ZS14]. Augot et al. [ALS14] give a storage-efficient PIR scheme where the servers only have to hold a part of the database. The 2-server PIR scheme of [DG15] has communication complexity of $\mathcal{O}(N^{(\frac{\log\log N}{\log N})^{\frac{1}{2}}})$ for a database with N entries.

Multi-Server PIR with Preprocessing. The bottleneck of many multi-server PIR constructions for large databases is the online computation time. For this reason, Beimel et al. [BIM00] proposed to include an offline preprocessing step that takes over some work that the server would process during the online phase. For instance, they design a preprocessing algorithm for Chor et al.'s protocol [CGKS95] that precomputes the XORs of some sub-cubes if the database is considered as a 3-dimensional cube. The server has to combine only the necessary stored values for the query which are in average much less XOR operations than XORing all the entries in the query. They also prove that every secure multi-server PIR protocol needs to access every entry in the database for each query without using preprocessing.

Demmler et al. [DHS17] also adapt this idea for their RAID-PIR scheme [DHS14] where each combination of XOR for subsets of a specific length are computed and stored in a preprocessing step using the *Method of four Russians* [ADKF70] for fast matrix multiplications and a Gray code to precompute these values efficiently.

These two previous works only spend a one-time offline effort to precompute database-dependent values that reduces the online computation to a constant number of XOR operations. Our scheme additionally spends multiple times effort to precompute query-dependent preprocessing values by letting the servers choose a part of the client's query. Thus, our scheme reduces the number of XOR operations significantly compared to Beimel et al.'s [BIM00] and Demmler et al.'s [DHS17] optimizations.

Single-Server PIR. Kushilevitz and Ostrovsky [KO97] showed that PIR can also be achieved with a single server under computational assumptions. They use a recursive process that partitions the database in every query until the recursion base is reached. However, [CMO00] proved that single-server PIR implies Oblivious Transfer (OT), i.e., we cannot get single-server PIR solely from one-way functions due to the separation result of [IR89]. Olumofin and Goldberg [OG11] showed in their experiments that the lattice-based single-server PIR scheme of [MG08] can be more efficient than downloading the whole database.

Another approach for single-server PIR is homomorphic encryption like the scheme of [KLL$^+$15; LP17], which achieves a rate of $(1-o(1))$. The communication rate of a PIR scheme is defined as $(\log_2 N + l)/L$ where l denotes the bitwidth of each element of the N database entries and L is the communication of the protocol. This scheme is not yet practical for large-scale databases since it is based on the homomorphic encryption scheme of [DJ01] that requires the server to compute a modular exponentiation for each bit of the database. Recently, Gentry and Halevi [GH19] developed a compressible fully homomorphic encryption scheme that can be used for efficient single-server PIR on large-scale databases. They conclude in their analysis that an implementation of their scheme should be faster than downloading the whole database and the server's computation measured in clock cycles could even be faster than encrypting the database with AES-NI.

2 Preliminaries

In this chapter we introduce basic notations, definitions and concepts used throughout this thesis. Our notations are given in Sect. 2.1. In Sect. 2.2, we briefly define Private Set Intersection (PSI) that is used by [TPY$^+$19]'s Google Password Checkup (GPC) protocol shown in 4.2 and by us for our new Compromised Credential Checking (C3) protocol presented in Sect. 4.3. We introduce the *Method of four Russians* [ADKF70] in Sect. 2.3 that is used by Demmler et al. [DHS17] for optimizing the RAID-PIR scheme [DHS14].

2.1 Notations

In this section, we give our basic notations and definitions used throughout this thesis. Tab. 2.1 shows the variables used for describing Private Information Retrieval (PIR) schemes in Chapt. 3, and Tab. 2.2 extends and overwrites the variables used for C3 in Chapt. 4.

We write $[n]$ for the set $\{0,\dots,n-1\}$ and $[x;y]$ for the set $\{x,x+1,\dots,y-1\}$ for $x<y$. The XOR of bit a and bit b is denoted by $a \oplus b$ and is 1 iff $a \neq b$.

2.2 Private Set Intersection (PSI)

Private Set Intersection (PSI) is a cryptographic protocol that allows two parties to compute the intersection of their private inputs X and Y. At the end of the protocol both parties only learn the intersection $X \cap Y$ of their inputs. The first construction was proposed by Meadows in [Mea86] and is based on the Diffie-Hellman key exchange.

[KLS$^+$17] optimize four existing PSI protocols for the special case where the input set of one party contains only a few elements (e.g., 1024 elements) while the set of the other party is very large (e.g., millions of elements). Two of their protocols are further improved and implemented on mobile clients by [KRS$^+$19].

Example applications for PSI with unequal set sizes are contact discovery on smartphones and discovering leaked passwords. The first application was integrated in the open-source messenger Signal as proof-of-concept by [KRS$^+$19]. In the latter case one party holds a set that contains only one element (its password) while the other party holds a set of leaked passwords with millions of elements. For example, Thomas et al.'s GPC tool [TPY$^+$19] has a set of 2.2 billion leaked username and password combinations.

Variable	Definition
D	raw data, i.e., a sequence of 0 and 1
$D[i]$	i-th entry of D, usually each entry has the same length
D_i	i-th block of D, where each block contains a specific number of entries
d	specific data entry in D
N	number of data entries
DB	a database, that builds a logical structure about raw data D
DB_i	part of the database that is stored by server i
M_i	memory for precomputations of server i
Q_i	queue for (seed, value)-pairs of server i
n	number of servers
idx	index of the data entry $D[idx]$ or the block B_{idx} the clients wants to retrieve
q_i	PIR query for server i
a_i	PIR answer of server i
b	size of a block
B	number of blocks in which D is separated
B_i	block number i
c	number of chunks
C_i	chunk number i
$flip_i$	flip chunk for server i
k	number of blocks in each chunk
r	redundancy parameter, i.e., the number of chunks each server has to process
e_i^j	j bit zero string with a 1 at position i
κ	security parameter
s_i	κ bit seed for server i generated on client side
S_i	κ bit seed for server i generated on server side
A_i	precomputed value of server i
t	size of the groups for precomputation

Table 2.1: Notations for Private Information Retrieval (PIR).

Variable	Definition
u	username
p	password
a	client's private key for PSI
b	server's private key for PSI
H	hash value
H^a	hash value blinded with the client's private key a
H^b	hash value blinded with the server's private key b
H^{ab}	hash value blinded with client's and server's keys a and b
z	prefix parameter, i.e., the number of bytes indicating the block position
P	specific block / partition of the database DB

Table 2.2: Notations for Compromised Credential Checking (C3).

2.3 Method of four Russians

The *Method of Four Russians* [ADKF70] refers to an algorithm with complexity $\mathcal{O}((\log d)(v^3/\log v))$ that is able to find the transitive closure of a directed graph having v nodes and diameter d.[1] Such a graph can be represented by an adjacency matrix that specifies for each pair of nodes if they are connected by an edge. The problem of finding the transitive closure of a graph is equivalent to exponentiating the boolean adjacency matrix. So, the method of four Russians can be used for matrix squaring and a matrix multiplication algorithm can be derived from the original algorithm. Demmler et al. [DHS17] use this matrix multiplication algorithm for improving their RAID-PIR scheme [DHS14] as described in Sect. 3.2.

Let us fix a $a \times b$ binary matrix A and a $b \times c$ binary matrix B. To compute the $a \times c$ matrix C we can split A into b/t groups $A_0, \ldots, A_{b/t-1}$ of t columns, and B into b/t groups $B_0, \ldots, B_{b/t-1}$ of t rows. Then $C_i = A_i B_i$ is an $a \times t$ times $t \times c$ matrix multiplication resulting in an $a \times c$ matrix for $i = 0, \ldots, t-1$ as also described in [Bar09]. We can derive the matrix C as follows:

$$C = AB = \sum_{i=0}^{b/t-1} A_i B_i = \sum_{i=0}^{b/t/1} C_i. \tag{2.1}$$

Arlazarov et al. [ADKF70] make a Gray code table M^i containing all 2^t linear combinations for matrix B_i for group $i = 0, \ldots, t-1$. The entries in the Gray Code table are ordered s.t. neighbors online differ by exactly one bit, e.g., a Gray Code over 3 bit is the sequence $(000, 001, 011, 010, 110, 111, 101, 100)$. The g-th entry in a Gray code table can be computed as $g \oplus (g >> 1)$. The advantage of using Gray codes for precomputing the linear

[1]A graph's diameter specifies the longest shortest path between any two nodes.

combinations is that each entry in the table can be computed by adding one t-bit value to the previous computed linear combination, since neighboring Gray codes only differ in one bit. If bit j of the next Gray code is flipped, we can add the j-th entry of B_i to the previous linear combination to compute the correct sum.

For computing AB, we initialize the $a \times c$ matrix C with zeroes and divide it into t $b/t \times c$ sub-matrices $C_0, \dots, C_{t-1}$. Let g_j be the t-bit binary number of row j of matrix A. We can add M^{i,g_j} to sub-matrix C_j for $i = 0, \dots, b/t - 1$ and $j = 0, \dots, a - 1$, where M^{i,g_j} denotes the entry of M^i with Gray code g_j.

3 Private Information Retrieval Optimizations

Private Information Retrieval (PIR) describes a protocol between a client and n servers that allows the client to securely query data in a public database DB held by the servers. At the end of the protocol the servers should learn neither the query nor the data that the client receives, while the client learns the queried entries. It is necessary that some of the servers are non-colluding. The idea and first construction for PIR was given in [CGKS95]. We focus on so-called *Multi-Server PIR* schemes since they have lower computational overhead that is valuable for mobile clients. In multi-server PIR the database DB is split over $n \geq 2$ non-colluding servers and the client sends a request to each of the servers.

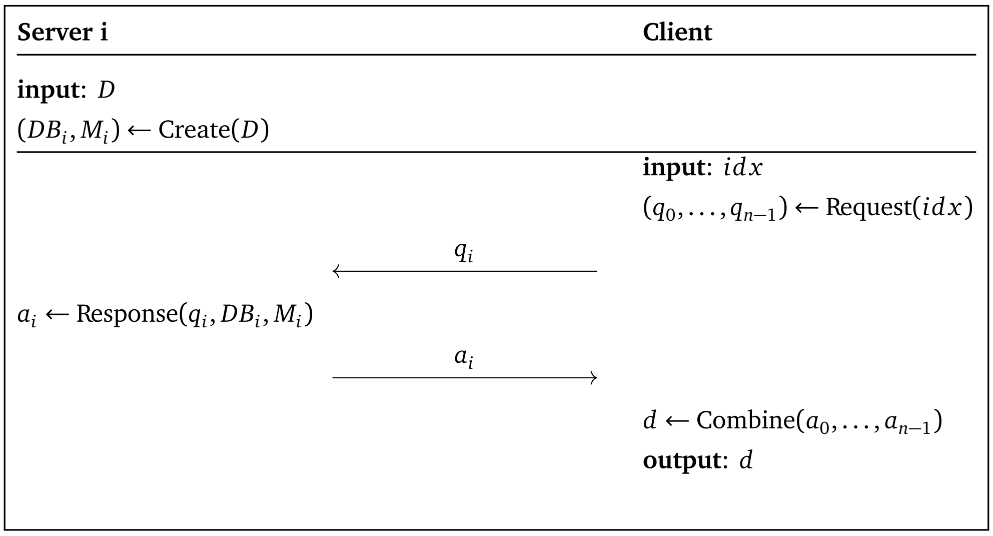
3.1 Our PIR Model

We define a classical PIR protocol as a tuple of algorithms (Create, Request, Response, Combine) as summarized in Prot. 3.1 and described below:

Create takes as input some data D and outputs the tuple (DB_i, M_i) for each server $i \in [n]$. A database DB is generated from the data D and a unique part of the database denoted as DB_i is sent to server i. Additionally, the Create algorithm may allocate memory M_i for each server i that is used for some precomputations which depend only on DB like in the RAID-PIR scheme [DHS14; DHS17]. This step is done one time for setting up the system without any communication to the client.

Request takes as input the index idx of the data to access and outputs a query q_i for each server i. Response is called on each server. It takes the query q_i as input and outputs an answer a_i based on the local database DB_i and precomputed memory M_i. The client collects the answers $a_0, \ldots, a_{n-1}$ of each server and calls the Combine algorithm that outputs the desired data entry d.

A PIR scheme is called *secure* if any combination of less than $t < n$ servers does not learn any information about the index idx or the requested data $d = D[idx]$ from their combined view of queries q_i. A PIR scheme is called *correct* if $D[idx] = \text{Combine}(\text{Response}^0(\text{Request}(idx)[0]), \ldots, \text{Response}^{n-1}(\text{Request}(idx)[n-1]))$, where Response^i denotes the function call of server i.


Server i
Client
input: D
$(DB_i, M_i) \leftarrow \text{Create}(D)$
input: idx
$(q_0, \ldots, q_{n-1}) \leftarrow \text{Request}(idx)$
q_i
$a_i \leftarrow \text{Response}(q_i, DB_i, M_i)$
a_i
$d \leftarrow \text{Combine}(a_0, \ldots, a_{n-1})$
output: d

Protocol 3.1: Message flow for a classical PIR protocol. The protocol is shown for server i with $i \in [n]$. Note that the client communicates with all n servers.

3.2 RAID-PIR [DHS14]

The multi-server PIR scheme RAID-PIR was developed by Demmler et al. [DHS14] and further improved in [DHS17]. RAID-PIR uses ideas from Redundant Array of Inexpensive Disks (RAID) storage systems like distributing the data over multiple servers to improve performance. As the disks in RAID, the servers only have to store parts of the data which then are combined using XOR operations. We give the pseudocode for the algorithms in our PIR model from Sect. 3.1 in Listing 3.2.1 and explain more details next.

The offline method Create splits the input data D into B blocks of size b each denoted as $B_0, \ldots, B_{B-1}$, i.e., D has Bb bits (line 4 in Listing 3.2.1a). These blocks are again split into $c \leq B$ chunks $C_0, \ldots, C_{c-1}$, where $C_i = B_{ki} || \ldots || B_{ki+k-1}$ for $i \in [0; r[$ and $k = B/c$ denotes the number of blocks for each chunk (lines 8 - 10 in Listing 3.2.1a).[1]

We define the whole database DB as $DB = C_0 || \ldots || C_{c-1}$. However, only a subset of the DB is sent to each server, denoted as DB_i. A redundancy parameter r with $2 \leq r \leq c$ gives us the number of chunks that each server has to process per query. The getChunks function from line 12 in Listing 3.2.1a returns the r chunks that server i has to touch. Concretely, getChunks(i, C_0, ..., C_{c-1}) returns the chunks C_i, $C_{i+1 \bmod c}$, $\cdots$, $C_{i+r-1 \bmod c}$. M_i is set to $\perp$ since there is no precomputation in the original RAID-PIR scheme by [DHS14] (line 13 in Listing 3.2.1a). We describe the precomputation improvements of [DHS17] in Sect. 3.2.2.

[1]For simplification, we assume that $c = n$ that is a realistic setting. We note that $c > n$ is only considered for the special case of multi-query RAID-PIR. In this case, the client can access up to c database entries within one query. We describe this in more detail in Sect. 3.2.1.

RAID-PIR without Preprocessing

```
// raw data $D$
function Create($D$)
    // split $D$ into $B$ blocks
    $(B_0, \dots, B_{B-1}) \leftarrow D$
    Let $c$ be the number of chunks
    $k \leftarrow \frac{B}{c}$
    // split blocks into $c$ chunks
    for $i \in [c]$ do
        $C_i \leftarrow B_{ki} || \dots || B_{ki+k-1}$
    endfor
    for $i \in [n]$ do
        $DB_i \leftarrow$ getChunks($i, C_0, \dots, C_{c-1}$)
        $M_i \leftarrow \perp$
    end for
    // database $DB_i$
    // precomputations $M_i$
    return $((DB_0, M_0), \dots, (DB_{n-1}, M_{n-1}))$
end function
```

(a) Create

```
// database $DB_i$
// precomputations $M_i$
// query $q_i$
function Response($DB_i, M_i, q_i$)
    parse $q_i$ to $(flip_i, s_i)$
    $q \leftarrow flip_i || \mathrm{PRG}(s_i)$
    $a_i \leftarrow q \cdot DB_i$
    // answer $a_i$ of server $i$
    return $a_i$
end function
```

(b) Response

```
// index $idx$
function Request($idx$)
    // generate queries for each server and chunk
    for $i \in [n]$ do
        $s_i \leftarrow_{\$} \{0,1\}^{\kappa}$
        for $j \in [c]$ do
            $q_i^j \leftarrow$ generateSubQuery($s_i, i, j$)
        end for
    end for
    // combine all queries and flip the $idx$−th bit
    $Q \leftarrow \bigoplus_{i=0}^{n-1} q_i^0 || \dots || q_i^{c-1}$
    $(flip_0 || \dots || flip_{c-1}) \leftarrow Q \oplus e_{idx}^B$
    for $i \in [n]$ do
        $q_i \leftarrow (flip_i, s_i)$
    end for
    // query $q_i$ for server $i$
    return $(q_0, \dots, q_{n-1})$
end function
```

(c) Request

```
// answers $a_0, \dots, a_{n-1}$
function Combine($a_0, \dots, a_{n-1}$)
    $d \leftarrow \bigoplus_{i=0}^{n-1} a_i$
    // desired data block $d$
    return $d$
end function
```

(d) Combine

Listing 3.2.1: Pseudocode for the four RAID-PIR algorithms [DHS14] without precomputations.

After setting up the system the client can call the Request method on input idx to retrieve the block B_{idx}. The idea of RAID-PIR is that each server computes XOR operations on a

q_0	011010	100010	011001	
q_1		101101	010110	100001
q_2	001101		001101	101001
q_3	010111	001011		001000
e_9^{24}	000000	000100	000000	000000

Figure 3.1: Example RAID-PIR queries with $n = 4$ servers, $B = 24$ blocks, $c = 4$ chunks, chunk size $k = 6$ and redundancy parameter $r = 3$. The orange cells are the flip chunks for the servers while the white cells contain the random sub-queries. The client requests the block with index 9.

subset of their stored blocks and sends the result back to the client. The client computes the XOR on all received results and obtains the block B_{idx}. The subset of blocks that each server touches is determined by the query that the client sends. In Chor et al.'s protocol[CGKS95], the client samples a random B-bit query q_i for servers $i = 0, \dots, n-2$ while the query q_{n-1} for server $n-1$ cancels out all unwanted blocks with $q_{n-1} = e^B_{idx} \oplus q_0 \oplus \dots \oplus q_{n-2}$, where e^B_{idx} is the B bit zero vector with a one at position idx.

In RAID-PIR the query that cancels out all unwanted blocks is distributed over all n servers. The first chunk that each server i stores, namely C_i, is the so-called *flip-chunk*. This flip chunk cancels out all unwanted blocks within this chunk. So, the client generates a $(r-1)k$-bit random query for each server i and computes the query for their flip-chunk denoted as $flip_i$. A small example is given in Fig. 3.1.

Instead of sending the whole random part of the query to server i, the client randomly samples a κ-bit seed s_i (line 5 in Listing 3.2.1c), where $\kappa = 128$ is the security parameter that holds $\kappa \ll (r-1)k$ for large databases.

The algorithm Request achieves this goal as follows. It generates a random seed s_i for each server i (line 5 in Listing 3.2.1c). With this seed, we compute the sub-query q_i^j for each chunk j for server i by calling generateSubQuery(s_i, i, j). This function gives us a k-bit zero string if chunk j is not processed by server i or if $i = j$. In all other cases a k-bit string is returned determined by passing s_i to a pseudorandom generator PRG, i.e., $(q_i^{i+1 \bmod c}, \dots, q_i^{i+r-1 \bmod c}) = \mathrm{PRG}(s_i)$.

In line 11 in Listing 3.2.1c, we set $Q = \bigoplus_{i=0}^{n-1} q_i^0 || \dots || q_i^{r-1}$, which gives us the accumulated random query so far. By computing $Q \oplus e^B_{idx}$ we get the flip chunk $flip_i$ for each server i as in line 12 in Listing 3.2.1c. Each server i receives the query $q_i = (flip_i, s_i)$ (lines 13 - 15 in Listing 3.2.1c).

For the Response method, each server i calls PRG(s_i) with s_i from q_i and retrieves $q_i^{i+1 \bmod c}, \dots, q_i^{i+r-1 \bmod c}$. Each q_j^i $[flip_j]$ is a k-dimensional vector while the corresponding C_j is a $k \times b$ matrix with rows $B_{kj}, \dots, B_{kj+k-1}$. By multiplying $a_i^j = q_i^j C_j$ $[a_i^j = flip_i C_j]$ we retrieve the

q_0	010010	100010	011001	
q_1		101101	010110	100001
q_2	001101		001100	101001
q_3	010111	001011		011000
e	001000	000100	000001	010000

Figure 3.2: Example Multi-Query RAID-PIR queries with $n = 4$ servers, $B = 24$ blocks, $c = 4$ chunks, chunk size $k = 6$ and redundancy parameter $r = 3$. The orange cells are the flip chunks for the servers while the white cells contain the random sub-queries. The client requests the blocks with indices 2, 9, 17 and 19.

answer of server i as $a_i = (a_i^i, \ldots, a_i^{i+r-1 \bmod c})$.[2] Then, $a_i = \bigoplus_{j=0}^{r-1} a_i^j$ is the answer of server i. Instead of processing one matrix multiplication for each chunk, we create a (kr)-dimensional vector q out of $flip_i$ and PRG(s_i) in line 6 in Listing 3.2.1b, consider DB_i as a $(kr) \times b$ matrix and compute $a_i = q \cdot DB_i$ (lines 6 - 7 in Listing 3.2.1b).

At the end of the protocol the client calls the Combine method that computes $d = \bigoplus_{i=0}^{n-1} a_i$ to obtain $d = B_{idx}$.

3.2.1 Multi-Query RAID-PIR [DHS14]

As already mentioned, RAID-PIR allows multi-query PIR, i.e., a client can receive up to c blocks within one query under the condition that the queried blocks are chunk-disjoint. Fig. 3.2 adapts the example from Fig. 3.1 to the multi-query case.

If the client requests the blocks with indices $idx_0, \ldots, idx_{c-1}$, line 12 of the Request algorithm in Listing 3.2.1c has to be modified as follows:

$$(flip_0||\ldots||flip_{c-1}) \leftarrow Q \oplus e^B_{idx_0} \oplus \ldots \oplus e^B_{idx_{c-1}}. \tag{3.1}$$

In the Response method in Listing 3.2.1b, server i computes the values a_i^j as described above and sends them all to the client. The client then computes for each chunk j the output $d_j = \bigoplus_{i=0}^{n-1} a_i^j$.[3]

If the client only retrieves one block, we have r times more communication from the server to the client in the multi-query variant of the protocol since the servers send a randomized block for each of their chunks. However, if the client queries multiple blocks, the multi-query variant is more efficient when the requested blocks are distributed equally along the chunks. We analyze and compare concrete communication and computation complexities in Sect. 3.5.

[2]Note that the matrix multiplication is on bit-level, i.e., multiplications and additions are AND and XOR operations.

[3]Some a_i^j are undefined since not every server processes all chunks. We set these undefined values to 0^b without loosing correctness.

3.2.2 RAID-PIR with Preprocessing [DHS17]

While the communication of the RAID-PIR protocol is very efficient, the vector-matrix multiplications on the server side are the bottleneck of the protocol. Therefore, Demmler et al. [DHS17] use the *Method of four Russians* [ADKF70] to precompute some parts of the results.

They propose that in addition to the RAID-PIR scheme without preprocessing shown in Listing 3.2.1, the Create algorithm assigns the memory M_i the returned value of algorithm FourRussians(DB_i) shown in Listing 3.2.2 and the Response algorithm is replaced as shown in Listing 3.2.2.

The FourRussians algorithm splits the database DB_i into groups of size t. This is done by splitting each chunk j of the server's r chunks into $\lceil k/t \rceil$ groups (line 7 in Listing 3.2.2a). The idea is that each of the 2^t combinations of each group is precomputed, i.e., the server just has to read the precomputed data for a group instead of XORing all t blocks in the group depending on the query. For reducing computation Demmler et al. [DHS17] use Gray codes for precomputing these combinations. The advantage of Gray codes is that two successive codewords in the Gray code only differs in only one bit, i.e., the corresponding precomputed values also only differ in only one block. So, we only have to compute one XOR operation on b bits for computing one of the $r \left\lceil \frac{k}{t} \right\rceil 2^t$ entries in the memory M_i.

The Gray code for an element g is the value $g \oplus (g >> 1)$ (line 15 in Listing 3.2.2a). Offset o gives us the index of the block in which the current Gray code and the previous Gray code differ. The precomputed value at the Gray code's position in the current group z in chunk j is the last precomputed value XOR the o-th block in this group (line 18 in Listing 3.2.2a). We denote this precomputed value as $M_i^{j,z,graycode}$ where $graycode$ is the current Gray code.

The Response algorithm also splits the client's query in groups of size t (lines 6 and 11 in Listing 3.2.2b).[4] We initialize the answer a_i to 0^b and process XOR operations step by step with a subset of the precomputed values in M_i. Concretely, we choose exactly one of the 2^t precomputed values in M_i for each group depending on the corresponding sub-query (line 13 in Listing 3.2.2b). For each chunk, the server now only has to compute $\lceil t/n \rceil - 1$ XOR operations instead of k XORs in [DHS14]. This time, the server i sets $a_i^j = \bigoplus_{m=0}^{\lceil k/t \rceil - 1} M_i^{j,m,q_i^{j,m}}$, i.e., the server only has to compute $\lceil t/n \rceil - 1$ XOR operations instead of k XORs for each chunk. The efficiency of the Response method grows linearly with t while the size of the memory M_i grows exponentially.

An advantage of preprocessing RAID-PIR is that the number of XOR operations is constant for each query. This prevents Denial of Service (DOS) attacks where an attacker can generate a worst case scenario in which a server has to perform a huge number of XOR operations.

[4] Firstly, the server extracts the query for each chunk in line 6 in Listing 3.2.2b and splits the extracted sub-queries into groups of size t.

RAID-PIR with Preprocessing

```
// database $DB_i$
function FourRussians($DB_i$)
  Let $t$ be the groupsize
  $groups \leftarrow \lceil k/t \rceil$
  // redundancy parameter $r$
  for $j \in [r]$ do
    $(C_{j,0}, \ldots, C_{j,groups-1}) \leftarrow C_j$
    for $z \in [groups]$ do
      $lgc \leftarrow 0$ // last Graycode
      $gc \leftarrow 0$ // current Graycode
      $diff \leftarrow 0$ // Graycode difference
      $M_i^{j,z,0} \leftarrow 0^b$
      for $g \in [1; 2^t]$ do
        $lgc \leftarrow gc$
        $gc \leftarrow (g \oplus (g >> 1))$
        $diff \leftarrow gc \oplus lgc$
        Let $o$ be the position of the 1 in $diff$
        $M_i^{j,z,gc} \leftarrow M_i^{j,z,lgc} \oplus C_{j,z}[o]$
      end for
    end for
  end for
  // precomputations $M_i$
  return $M_i$
end function
```

(a) FourRussians

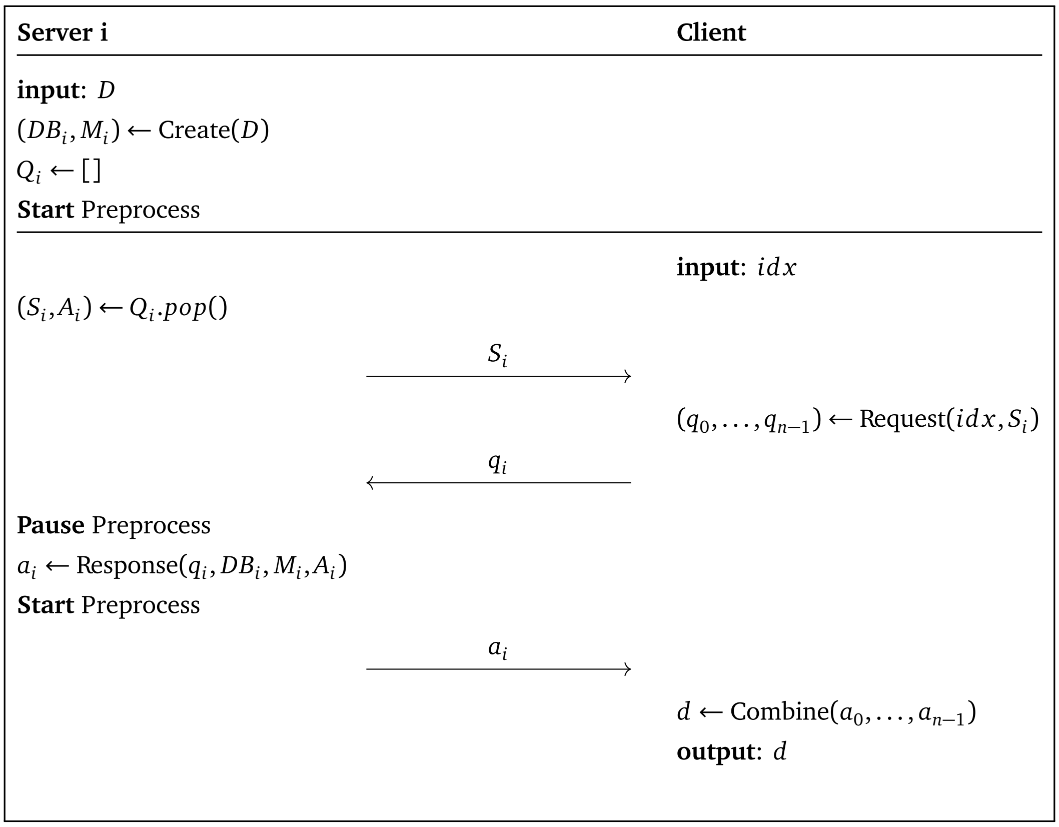
```
// database $DB_i$
// precomputations $M_i$
// query $q_i$
function Response($DB_i, M_i, q_i$)
  parse $q_i$ to $(flip_i, s_i)$
  $(q_i^0, \ldots, q_i^{r-1}) \leftarrow flip_i || \mathrm{PRG}(s_i)$
  Let $t$ be the groupsize
  $groups \leftarrow \lceil k/t \rceil$
  $a_i \leftarrow 0^b$
  for $j \in [r]$ do
    $(q_i^{j,0}, \ldots, q_i^{j,groups-1}) \leftarrow q_i^j$
    for $z \in [groups]$ do
      $a_i \leftarrow a_i \oplus M_i^{j,z,q_i^{j,z}}$
    end for
  end for
  // answer $a_i$ of server $i$
  return $a_i$
end function
```

(b) Response

Listing 3.2.2: Pseudocode for the algorithm FourRussians and the modified Response algorithm for the RAID-PIR scheme with precomputation.

3.3 Query-Dependent Preprocessing

Previous works like [BIM00; DHS17] use preprocessing to reduce the online computation for the server by merging parts of the database into groups in a one-time offline phase, s.t. the server has to compute a constant number of XOR operations online. During the Response method, the servers only have to combine the precomputed parts depending on the query q_i. Our idea is to split the preprocessing into two parts - the *Database Preprocessing* and the *Per-Query Preprocessing*. The database preprocessing is a one-time precomputation step that maps the database into a state that enables the servers to compute their answer more quickly as described in Sect. 3.2 and in [BIM00; DHS17]. In addition, the per-query preprocessing is a routine that precomputes concrete parts of the server's answer in a separate process, i.e., it is a continuous process that only pauses if the reserved memory space for computed concrete parts of the server's answer is full or under certain conditions, e.g., if the server receives a query, the server uses its whole computation power to process the query and therefore pauses the Preprocess routine. In Chapt. 5, we explain and discuss some possible conditions. We define our new preprocessing model in Sect. 3.3.1.



Server i		**Client**
input: D		
$(DB_i, M_i) \leftarrow \text{Create}(D)$		
$Q_i \leftarrow []$		
Start Preprocess		
		input: idx
$(S_i, A_i) \leftarrow Q_i.pop()$		
	$\xrightarrow{S_i}$	
		$(q_0, \ldots, q_{n-1}) \leftarrow \text{Request}(idx, S_i)$
	$\xleftarrow{q_i}$	
Pause Preprocess		
$a_i \leftarrow \text{Response}(q_i, DB_i, M_i, A_i)$		
Start Preprocess		
	$\xrightarrow{a_i}$	
		$d \leftarrow \text{Combine}(a_0, \ldots, a_{n-1})$
		output: d

Protocol 3.2: Message flow for a query-dependent preprocessing PIR protocol. The protocol is shown for server i with $i \in [n]$. In this variant, the servers pauses the Preprocess algorithm so that they can use there whole computation power for the Response algorithm. Note that the client communicates with all n servers.

3.3.1 Query-Dependent Preprocessing Model

A PIR scheme in the query-dependent preprocessing model is a tuple of algorithms (Create, Preprocess, Request, Response, Combine). The protocol is shown in a high-level in Prot. 3.2.

As in our model from Sect. 3.1, the Create algorithm takes some input data D and outputs the tuples (DB_i, M_i) for each server $i \in [n]$. A database DB is generated from the data D and a unique part of the database denoted as DB_i is sent to server i. Additionally, the Create algorithm may allocate memory M_i for each server i that is used for some precomputations which depend only on DB. This step is done one time for setting up the system without any communication to the client.

Each server runs the Preprocess algorithm locally in parallel in a separate thread that can be started and paused. It takes as input the database DB_i and the memory space for precomputations M_i, adds query-specific tuples (S_i,A_i), and stores them in the queue Q_i until it is full or the thread is interrupted. Afterwards it is paused until it is again started when there is new space for more values in Q_i.

Request takes as input the index idx of the data to access in addition to some seeds $S_1,\dots,S_n$ from the n servers that contain information for creating suitable queries $q_1,\dots,q_n$. A query is suitable if it helps the servers to save computation time by using data from the queue Q_i.

Each server i calls the Response algorithm on input DB_i, M_i, A_i and the received query q_i. Within this method the servers compute their answers a_i.

In the last step, the client collects all the server answers $a_i,\dots,a_n$ and calls the Combine algorithm that gives the requested data entry $d = D[idx]$.

3.3.2 RAID-PIR with Query-Dependent Preprocessing

In the following, we extend the RAID-PIR scheme [DHS14; DHS17] into a PIR scheme in the query-dependent preprocessing model. The Create and Combine algorithms are identical to the original RAID-PIR scheme with the preprocessing optimization described in Sect. 3.2 and shown in Listing 3.2.1. In contrast to RAID-PIR [DHS14; DHS17], the servers choose the seed for generating their random sub-queries. For this seeds, they already compute the necessary XOR operations and store the results locally. When the servers receive a query, they send the precomputed seed to the client, who collects all the seeds and generates the flip chunk queries for the servers. After the servers receive the query for their flip chunks, they take the precomputed data related to the previous sent seed and only compute the remaining XOR operations specified in the flip chunk query.

The Request algorithm is slightly changed to the one presented in Listing 3.2.1. It takes - aside from the index idx - the seeds $S_0,\dots,$ S_{n-1} from the servers as inputs. In line 5 in Listing 3.2.1c on page 12, we do not sample the seed anymore since the server did that already, i.e., we set $s_i = S_i$. Since the servers already know their seeds, the client does not need to send it back again, so we set $q_i = flip_i$ in line 14 in Listing 3.2.1c.

Pseudocodes of the remaining two algorithms Preprocess and Response are given in Listing 3.3.1. Preprocess is the new algorithm of this RAID-PIR scheme in the query-dependent preprocessing model that is called by each server i on inputs DB_i, M_i and Q_i. Server i

samples a random κ-bit seed S_i (line 8 in Listing 3.3.1a) to generate the pseudo-random sub-queries for all of its non-flip chunks (line 9 in Listing 3.3.1a). With these sub-queries the server can XOR the blocks of all non-flip chunks specified in the sub-queries similar to the Response method of the preprocessing RAID-PIR scheme described in Sect. 3.2 and shown in Listing 3.2.2. In line 19 in Listing 3.3.1a, we push the seed-value pair (S_i, A_i) to the queue Q_i and repeat all these steps until Q_i is full.

The Response algorithm splits the received query for the flip chunks into groups of size t (line 9 in Listing 3.3.1b). We initialize the answer a_i with the precomputed value A_i (line 8 in Listing 3.3.1b) and process XOR operations step by step with a subset of the precomputed values in M_i similar to the Response algorithm from the preprocessing RAID-PIR scheme described in Sect. 3.2 and shown in Listing 3.2.2.

It is important that the server discards the used seed after each protocol execution and never reuses it otherwise, the servers could launch the following attack: Assume a $n = 2$ server setting where server 0 has a constant set of seeds where it randomly chooses one for each query. A malicious server 1 could collect received queries by clients in a list L and computes for each new incoming query q the Hamming distances between q and the queries stored in L. Server 1 can then make some assumptions about the client's requesting data. The Hamming distance h between q and an entry in L denoted as q' can be interpreted as follows:

$h = 0$: It holds that $q = q'$, i.e., the client either requests the same entry in the server's flip chunk or some entry not included in its flip chunk.

$h = 1$: It is very likely that either q or q' is the query that completely cancels out the seed of server 0, i.e., the server can determine the block that one of the two queries accesses, namely the one where the two queries distinguish.

$h = 2$: It is very likely that q and q' both access blocks in the malicious server's flip chunks, namely these two blocks where the two queries differ.

$h > 2$: The server has not received a similar query yet, so we cannot make any assumption about the client's query.

The more data the server collects, the more fine-grained assumptions can it make about the client's query. This attack also works in a PIR scheme where more than two servers operate. The only requirement is that all servers have a fixed set of seeds they use. So, to prevent this attack, we let the servers discard every seed after using it.

3.4 Compressible PIR

Multi-server PIR schemes allow clients to securely retrieve information without letting the servers know what information the clients want to access. However, if the servers collude, they learn the desired data entry and the client's privacy is broken. Another drawback of multi-server PIR is the large amount of data the servers have to store, especially for the

RAID-PIR with Query-Dependent Preprocessing

```
// database $DB_i$
// precomputations $M_i$
// queue of (seed, value)-
    pairs $Q_i$
function Preprocess($DB_i, M_i, Q_i$)
  Let $t$ be the groupsize
  $groups \leftarrow \lceil k/t \rceil$
  while $Q_i$ is not full do
    $S_i \leftarrow_\$ \{0,1\}^\kappa$
    $(q_i^1, \ldots, q_i^{r-1}) \leftarrow \mathrm{PRG}(S_i)$
    $A_i \leftarrow 0^b$
    for $j \in [1;r]$ do
      $(q_i^{j,0}, \ldots, q_i^{j,groups-1}) \leftarrow q_i^j$
      for $z \in [groups]$ do
        $A_i \leftarrow A_i \oplus M_i^{j,z,q_i^{j,z}}$
      end for
    end for
    // seed $S_i$
    // value $A_i$
    push ($S_i$,$A_i$) to $Q_i$
  end while
end function
```

(a) Preprocess

```
// database $DB_i$
// precomputations $M_i$
// precomputed answer $A_i$
// query $q_i$
function Response($DB_i, M_i, A_i, q_i$)
  Let $t$ be the groupsize
  $groups \leftarrow \lceil k/t \rceil$
  $a_i \leftarrow A_i$
  $(q_i^{0,0}, \ldots, q_i^{0,groups-1}) \leftarrow q_i$
  for $z \in groups$ do
    $a_i \leftarrow a_i \oplus M_i^{0,z,q_i^{0,z}}$
  end for
  // answer $a_i$ of server $i$
  return $a_i$
end function
```

(b) Response

Listing 3.3.1: Pseudocode for the algorithm Preprocess and the modified Response algorithm for the RAID-PIR scheme with query-dependent preprocessing.

preprocessing. To improve the current situation, we transfer the database compression idea of Tamrakar et al. [TLP+17] to PIR and call this optimization *Compressible PIR*.

Compressible PIR encapsulates a PIR scheme and can therefore be applied to any PIR scheme that is based on blocks. [TLP+17] use a compression to decrease the size of a database for their private membership test scheme. They use a compression function ρ on the database, where their decompression function ρ^{-1} requires the knowledge of the whole database. Since the client only retrieves one block of the database and decompressing a whole database on server-side would be very inefficient, we apply the compression function ρ to each block of the database, i.e., the client does not need to know any information but the retrieved block to decompress the block. For a better compression, we increase the blocksize b. Thus, we gain less blocks while the blocks contain more information and are additionally compressed which leads to saving storage / memory. A side effect of compressible PIR is the high rate of multi-queries within one query, since there are much more entries in a block. However, it can be very unlikely that two desired entries are located in the same block. On the one hand, this leads to more communication from the servers to client since the blocksize b grows, but on the other hand, the communication from the client to the servers reduces since there are less blocks.

We gain the best communication complexity $\mathcal{C}(b)$ by setting the blocksize $\hat{b} = \sqrt{|D|/c}$, where $|D|$ is the size of the database and c denotes the number of chunks. The total communication for the RAID-PIR scheme [DHS14] with preprocessing [DHS17] as described in Sect. 3.2.2 and our query-dependent preprocessing scheme presented in Sect. 3.3 is given in Eq. (3.2), where k denotes the number of blocks in each chunk, B denotes the number of blocks and κ is the security parameter. A detailed derivation of Eq. (3.2) is given in Sect. 3.5.

$$\begin{aligned} \mathcal{C}(b) &= k + \kappa + b \\ &= \frac{B}{c} + \kappa + b \\ &= \frac{|D|}{bc} + \kappa + b \end{aligned} \tag{3.2}$$

One can easily show by derivation that $\mathcal{C}(b)$ has its local minimum at $\hat{b} = \sqrt{|D|/c} = \sqrt{|D|/n}$ if we assume $c = n$, where n denotes the number of servers.

For our measurements in Sect. 6.3, we use the compression scheme of [TLP+17] for each block. The idea is, that we first sort the whole database before we separate it into blocks. Assume, a block has the entries $(e_0, \dots, e_N)$. Since the database is sorted, successive entries are close to each other and thus we can store their differences instead of the whole entries themselves: $(e_0, e_1 - e_0, e_2 - e_1, \dots, e_N - e_{N-1})$. It is easy to see that the length of the differences is smaller than the length of the entries.

3.5 Communication and Computation Complexity of RAID-PIR Schemes

In this section, we analyze the communication, computation and storage complexity of the RAID-PIR scheme by Demmler et al. [DHS14], the improved version using preprocessing [DHS17], our improvement utilizing query-dependent preprocessing and the multi-query variants of all the schemes. A summary of the three complexities with concrete values is given in Tab. 3.1.

Setup. Since the multi-query variants are worse if the client only retrieves one block, we assume that the client wants to learn u blocks. For the multi-query schemes, we distinguish between the best case, average case and worst case scenario regarding the distribution of the u blocks the client wants to retrieve since this highly influences the performance of those schemes. So, for the multi-query schemes we denote with U the number of necessary queries to retrieve all u blocks. In the best case scenario, all blocks are distributed equally along the c chunks ($U = u/c$) while in the worst case scenario, all wanted blocks are in the same chunk ($U = u$). The average case assumes that the blocks are equally distributed over half of the blocks ($U = 2u/c$). Demmler et al. [DHS17] force best case scenarios by distributing large data d with $|d| > b$ over chunk-disjoint blocks. However, this only works for databases containing data entries that are larger than the blocksize, which does not hold for our Compromised Credential Checking (C3) scenario.

We determine the upload (from client to server) and download (from server to client) communication separately. The computation complexity is determined by the number of XOR operations during the protocol (i.e., we do not count the computations in Create and Preprocess) since they are the bottleneck of the online phases of the schemes. We also separate client and server computation.

RAID-PIR without Preprocessing (Sect. 3.2). The client sends a κ bit seed and a k bit query for each of the u blocks to each server, while each server sends a b bit answer to the client for each block. So, we have a very efficient upload of $un(k + \kappa)$ bits and download of unb bits. We explain the online computations for the server and the client in the following. First of, we have a different look at line 11 of the client's Request algorithm in Listing 3.2.1c on page 12. For each of the c chunks there are $(n - r)$ of the subqueries q_i^j zero bit vector of size k since every server only processes r chunks. As a consequence, we do not have to count the XOR operations over zero vectors since an optimized implementation would also not compute them (Note that we will assume this for the other schemes, too). So, we have $r - 1$ XOR operations over k bit each in line 11 in Listing 3.2.1c. One more XOR operation over B bits is processed in line 12 in Listing 3.2.1c. This are in total $u(c(r - 1)k + B) = u(B(r - 1) + B) = uBr$ XOR operations. In the Combine algorithm, the client additionally computes $n - 1$ XOR operations over b bits, i.e., the client has a total computation of $u(Br + (n - 1)b)$ XOR operations. The server XORs approximately half of the blocks in DB_i over b bits each, which are in total $urkb/2$ XOR operations. Each server holds r chunks of the database, which are rkb bits.

For the multi-query variant the upload is $U \cdot n(k + \kappa)$. Each server replies with r blocks per chunk for each query, so the client downloads $Uncb$ bits. The client and server computation is similar to the non-multi-query variant, namely $U(Br + (n-1)b)$ and $Urkb/2$ XOR operations, respectively.

RAID-PIR with Preprocessing (Sect. 3.2.2). The client's upload, download and computation complexity is equal to the RAID-PIR scheme without preprocessing. Only the server's computation and the necessary storage differs. For each chunk, the server XORs k/t blocks of size b, so this are in total $urkb/t$ XOR operations ($Urkb/t$ for the multi-query variant). While the server's online computation reduces by a factor of t, the storage increases exponentially by a factor of $\approx 2^t$. For each group, the server has to store 2^t blocks of size b each. There are in total ck/t groups, so the server has to store $2^t ckb/t$ bits.

RAID-PIR with Query-Dependent Preprocessing (Sect. 3.3.2). The upload reduces by κ bits per query since the server sends the seed in the query-dependent preprocessing RAID-PIR scheme. So, we have an upload of unk bits (Unk for the multi-query variant). As already mentioned, the κ bits are moved to the download, which are in total $un(\kappa + b)$ bits ($Un(\kappa + cb)$ for the multi-query variant). The client's computation is equal while the server's computation reduces extremely. Each server XORs k/t blocks only on its flip-chunk, i.e., the server processes ukb/t XOR operations (Ukb/t for the multi-query variant). The storage slightly increases compared to the RAID-PIR scheme with preprocessing since $(\kappa + b)$ bits are stored for each precomputed. If the server has a capacity ϵ for precomputed values, the total storage are $2^t ckb/t + \epsilon(\kappa + b)$ bits.

Scheme	Communication Complexity	Concrete Value
RAID-PIR [DHS14]	$un(k+\kappa+b)$	5 092
RAID-PIR P [DHS17]	$un(k+\kappa+b)$	5 092
RAID-PIR QDP [this thesis]	$un(k+\kappa+b)$	5 092

(a) Communication Complexity (Bytes)

Scheme	Computation Complexity	Concrete Value
RAID-PIR [DHS14]	$u(Br+(rk/2+n-1)b)$	1 606.5
RAID-PIR P [DHS17]	$u(Br+(ck/t+n-1)b)$	406.3
RAID-PIR QDP [this thesis]	$u(Br+(k/t+n-1)b)$	206.3

(b) Online Computation Complexity (thousand XORs)

Scheme	Storage Complexity	Concrete Value
RAID-PIR [DHS14]	rkb	3.2
RAID-PIR P [DHS17]	$2^t ckb/t$	102.4
RAID-PIR QDP [this thesis]	$\epsilon(\kappa+b)$	102.5

(c) Storage Complexity (MB)

Table 3.1: Summary of communication (a), computation (b) and storage (c) complexity of RAID-PIR [DHS14], RAID-PIR with preprocessing (P) [DHS17] and RAID-PIR with query-dependent preprocessing (QDP) (this thesis). We also give concrete values using our parameters from one of our benchmarks in Sect. 6.2. We choose $n = 2$ servers, redundancy parameter $r = 2$, blocksize $b = 1\,265$, $N = 100\,000$ database entries, which results in $B = 2\,530$ blocks, security parameter $\kappa = 16$ byte (= 128 bit), groupsize $t = 8$, $c = 2$ chunks, a query buffer with $\epsilon = 100$ elements, $k = B/c = 1265$, and $u = 1$ query.

4 Compromised Credential Checking (C3)

In this chapter, we investigate Compromised Credential Checking (C3) that allows users to learn if their credentials are published in a data breach. Data breaches have become a very important topic since the online presence of billions of people are affected. Billions of username-password pairs are published in plaintext as so-called Collection 1-5 [Gre19]. Most people are not aware of the security of their passwords since they believe that it is not worthwhile to attack them. So, it is very important that we have C3 tools that notify people if their credentials are leaked in a data breach. Additionally, the users should not have much effort to change their password otherwise they ignore the warnings.

We look at existing C3 tools and discuss their security in Sect. 4.1. In Sect. 4.2 we detail the complete protocol of Google's Chrome plugin Google Password Checkup (GPC) [TPY$^+$19]. We extend GPC in Sect. 4.3 with a Private Information Retrieval (PIR) scheme and provide the first C3 tool that achieves perfect anonymity for the user.

4.1 Compromised Credential Checking Tools

In the recent years, some C3 tools were developed. The most prominent APIs are HaveIBeen-Pwned (HIBP) [Hun19] and ENZOIC [ENZ16a]. Recently, Google published the Chrome extension GPC [TPY$^+$19] that checks in real-time and in a privacy-preserving manner if a username and password combination is published in a data breach, whenever the user inputs their credentials to a website. Li et al. [LPA$^+$19] design two C3 protocols called Frequency-Smoothing Bucketization (FSB) and ID-based Bucketization (IDB). We can distinguish four different types of queries for C3 tools.

Username. Firstly, one can query the username, i.e., the checkup tool looks if passwords were published in combination with a given username. However, this information does not necessarily tells the user if a password is attacked, rather notice the users that at least one password is published in combination with their username. In many cases, this might be enough information since most of the users reuse the same password for different domains (or even use the same passwords for all services). But often, the published password is an old one and consequently the user is not affected.

Domain. Similarly, we face the same problem if we query a specific domain that might be affected by data breaches. The breached password could also be an old one or the account could be created after the date of the breach. If the password is breached at a domain the user is not very active, they will not be notified about the breach and the same credentials for other services are affected.

Password. A safe option for detecting compromised passwords is querying passwords themselves even though this requires a lot of trust in the C3 tool. The advantage is that the user gets notified for all of their active accounts if their password is in a data breach. Thomas et al. [TPY+19] argue that this option is too over-cautious since an attacker still has problems to hijack an account if the user's username is not included in the breach. Most of the services ensure that attackers cannot brute-force the correct password of the users without much overhead.

Username + Password. The most user-friendly option is to query the combination of username and password. If the C3 tool returns a match, the user can be sure that their account is vulnerable. The drawback of this solution is that the credentials for a user can still be vulnerable if they use the same passwords for accounts with similar usernames, e.g., if a given username u with password p is part of a data breach, the user will not be notified for an account with username $u@domain.com$. In many cases, a simple search engine query suffices to learn the E-mail address of a user.

Thomas et al. [TPY+19] conclude that querying the combination of username and password is the best option due to the user-friendliness. So, their Chrome plugin GPC only supports this option. Querying only passwords is the safest option since it ensures that the user does not use passwords published in a data breach, but it can alert the user too often. In both cases, the user has to trust the C3 tool since highly sensitive information about the password is sent to a server. The IDB protocol by [LPA+19] also focuses on checking username-password pairs very similar to GPC with the difference that the query does not contain information regarding the password. They sort the username-password pairs into blocks, where the block index is given by the first bytes of the hash value of the username, rather than of the hash value of the username-password pair. Thomas et al. independently also suggest this protocol and consider to include this in their GPC plugin [TPY+19]. However, this requires the user to compute one more expensive hash operation and thus they do not include this optimization into their plugin.

ENZOIC offers all four options while HIBP renounce the username-password combination. The FSB protocol by [LPA+19] queries only passwords. In the following sections we summarize some details about most of the mentioned C3 tools or protocols. We give a complete description of the GPC protocol in Sect. 4.2 and skip the IDB protocol since it is almost identical to GPC.

4.1.1 HaveIBeenPwned (HIBP) [Shi19]

HIBP is an API that is used by many web services like the 1Password [Shi19] password manager. Firefox Monitor is a security tool that warns the users when they access a website that is affected by a data breach. It additionally allows users to query email addresses that are forwarded to HIBP. GitHub checks with the help of HIBP if the passwords of their users are compromised [Mat18].

HIBP use SHA-1 to store the passwords in their database. A client who queries a specific password firstly computes the SHA-1 hash of their password and sends the first 20 bits to the server. The server then returns a list of all the password hashes that begin with the same 20 bits in combination with the number of occurrence in data breaches. Finally, the client checks if his local computed hash is in the list.

Querying the domain just returns true or false and querying by username returns all the SHA-1 hashed passwords in the database belonging to the given username in combination with the number of occurrences.

4.1.2 ENZOIC / PasswordPing [ENZ16b]

PasswordPing is an API that was founded in 2016 - 2 years earlier than HIBP- and recently rebranded as ENZOIC. ENZOIC is used by, e.g., the password manager LastPass [ENZ16b].

In contrast to HIBP, ENZOIC allows to query the SHA-1 hash of a username that lightly protects the anonymity of the user. Querying the password works similar to HIBP but reveals the first 40 bits of the hashed password. A user has the highly non-recommended options to query the plaintext or the complete hash of the password.

The most interesting option of ENZOIC is querying by the combination of username and password. The client queries the username or the hash of the username u and receives a salt s. In the next round the client computes $\mathcal{H}(u,p,s)$ with their password p and sends the first 40 bits of this hash to ENZOIC who returns all matched values.

4.1.3 Google Password Checkup (GPC) [TPY+19]

Google provide an innovative Chrome plugin that alerts the users whenever they enter a username-password combination that has been published in a data breach. While HIBP and ENZOIC are secure against the servers (aside from the fact that the servers learn information about the hash value of the user's sensitive data), a malicious client can learn information about the credentials of other users since the hash values in the set of sent passwords are not blinded. Thomas et al. [TPY+19] protect their scheme against malicious clients by using Private Set Intersection (PSI) in their protocol. However, their plugin also sends the domain to the server after each protocol execution [Kuk19] which is not necessary for their protocol. We give a complete description of the protocol in Sect. 4.2.

4.1.4 FSB [LPA+19]

Li et al.'s [LPA+19] IDB protocol can not adapted to the querying by only passwords case. Since this option is more recommended for highly security-sensitive users, they introduce FSB that works great with the querying only passwords option. Their protocol provides more security than the password-only option of HIBP and ENZOIC in the sense that an attacker has less advantage in guessing the password of a user when having access to the protocol execution view. FSB goes away from the idea of assigning the credentials into the blocks having the index equal to the first bytes of the hashes. In FSB, passwords are assigned to blocks depending on their frequencies, i.e., more frequent passwords are assigned to many blocks. This hardens guessing the correct password for an attacker since the conditional probability of frequent passwords is made similar to those of less frequent passwords. More details about this protocol can be found in [LPA+19].

4.2 Google Password Checkup Protocol (GPC) [TPY+19]

Thomas et al. [TPY+19] develop the GPC protocol that consists of four algorithms, namely CreateDatabase, CreateRequest, CreateResponse and Verdict. The CreateDatabase algorithm is a setup algorithm that creates the database, similar to our PIR model described in Sect. 3.1. The remaining three algorithms are shown in Prot. 4.1, where the first part of the client belongs to the CreateRequest algorithm, the other party to the Verdictalgorithm and the server's part are the steps of the CreateResponse algorithm. Within this protocol, a Diffie-Hellman based PSI protocol is integrated, where the server holds a secret key b and the client samples a random key a for each request. In the rest of this section, we detail each of the four algorithms.

CreateDatabase. The CreateDatabase algorithm is a preprocessing step that only has to be run once by setting up the service and during database updates. The server holds a set of N username and password pairs $D = \{(u_1, p_1), \ldots, (u_N, p_N)\}$ that has to be stored in a database DB so that a client can securely check if their password is in D. Thomas et al. [TPY+19] achieve this by hashing each entry (u_i, p_i) with a collision-resistant hash function $\mathcal{H}$ to $H_i = \mathcal{H}(u_i || p_i)$. They use *Argon2* as hash function $\mathcal{H}$ that is also used as proof of work for the client to prevent Denial of Service DOS attacks. Thereafter the hash H_i is already blinded with the server's secret key $b \in \mathbb{G}$ for the integrated PSI protocol with H_i^b =Blind(H, b). Since the integrated PSI protocol requires a commutative blinding for double-blinded messages[1], Google's protocol uses the elliptic curve *NID_secp224r1*. A prefix parameter z splits DB into $(2^8)^z = 256^z$ blocks (similar to the RAID-PIR scheme by [DHS14] as described in Sect. 3.2). The block position for the blinded hash H_i^b is given by the first z bytes of H_i, i.e., H_i^b is stored in block $DB_{H_i[0;z]}$.

[1] Commutative blinding allows us to unblind a multiple blinded message in any order.

CreateRequest. When the client creates their request for the server they behave similar to the CreateDatabase method of the server. The client holds its username and password pair (u, p) and generates for each request a secret key a. This pair is hashed with the hash function $\mathcal{H}$ as $H = \mathcal{H}(u||p)$.[2] As before, the hash H is blinded with the clients private key a to H^a = Blind (H, a). The prefix parameter z is also known to the client so they are able to extract the first z bytes of H as $idx = H[0; z]$. In the end, the request is the pair (idx, H^a), i.e., the first z bytes of H are revealed to the server. Google's GPC protocol [TPY+19] sets $z = 2$ and achieve a B-anonymity of roughly $B \approx 60\,000$, i.e., the server learns in which of the B blocks the user's credentials is stored (or would be stored if they are not leaked).

CreateResponse. After the server receives the request (idx, H^a) from the client it transmits the set of all the values stored in DB_{idx}. The drawback in this step is that the server learns in which partition of the database the user's credentials are potentially stored as already mentioned. Aside from transmitting this set the server blinds H^a with its private key b to receive H^{ab} =Blind (H^a, b) and sends H^{ab} additionally to the client.

Verdict. In the last step the client finishes the PSI protocol and checks if their credentials are vulnerable. Therefore, the client unblinds the received value H^{ab} to H^b =Unblind (H^{ab}, a). Finally, the client is able to lookup the value H^b in the received database partition DB_{idx}. It is clear to see that the underlying blinding scheme has to be commutative which is satisfied by using elliptic curve Diffie-Hellman (ECDH).

4.3 Hiding Queries with PIR

Google's protocol [TPY+19] provides a privacy-friendly solution for the C3 problem. However, the user's anonymity is not completely protected since $z = 2$ bytes of the hashed credentials are leaked to Google. To circumvent this problem we use a query-dependent preprocessing PIR scheme as described in Sect. 3.3, that hides the z bytes from the server at the cost of efficiency. Thomas et al. [TPY+19] mention that PIR cannot be efficient enough for this application and therefore do not include in in their GPC protocol although this would protect the user's data completely. Their protocol runs in approximately 8.5 seconds and thus it is obviously more efficient than our C3 tools using PIR, where the PIR part alone takes roughly 13 seconds (see Sect. 6.3), but we protect the user's data perfectly. Li et al. [LPA+19] showed in a simulation that an attacker can guess 33% of user's passwords within at most 10 attempt (66% within 1 000 attempts) if they know the protocol transcript of GPC. If an attacker has no access to the protocol transcript, they can only guess 56% within 1000 attempts. While it is unlikely that an attacker is able to have 1 000 attempts for most of the services, 10

[2]Thomas et al.[TPY+19] and Li et al.[LPA+19] independently propose to use $\mathcal{H}(u)$ s.t. the server has no information about the user's password. However, since the username-password pair is hashed in the server's database, the client has to compute $\mathcal{H}(u)$ anyway.

Server		**Client**
input: username||password data D		
$b \leftarrow_{\$} \mathbb{G}$		
$DB \leftarrow \text{CreateDatabase}(D, b)$		
		input: username u, password p
		$\text{CreateRequest}(u, p)$:
		$a \leftarrow_{\$} \mathbb{G}$
		$H \leftarrow \mathcal{H}(u\|\|p)$
		$H^a \leftarrow \text{Blind}(H, a)$
	$\xleftarrow{H^a, idx = H[0;z]}$	
$\text{CreateResponse}(DB, b, H^a, idx)$:		
$H^{ab} \leftarrow \text{Blind}(H^a, b)$		
$P \leftarrow DB_{idx}$		
	$\xrightarrow{H^{ab}, P}$	
		$\text{Verdict}(a, H^{ab}, P)$:
		$H^b \leftarrow \text{Unblind}(H^{ab}, a)$
		return $H^b \in P$ // **true** iff $u\|\|p$ **in** D

Protocol 4.1: GPC protocol. The four algorithms CreateDatabase, CreateRequest, CreateResponse, and Verdict are covered in this overview.

attempts are a realistic setting for an attacker and thus the GPC protocol leaks too much information. It is easy to see that we achieve perfect anonymity instead of B-anonymity due to the fact that the server learns no information about the users data since every information related to the credentials are encrypted or blinded. Using our query-dependent preprocessing RAID-PIR scheme, we need multiple non-colluding servers. Thomas et al. [TPY$^+$19] argue that a multi-server PIR scheme is not realistic for this application since it is hard to find two or more trustworthy servers. However, today it is feasible to find two non-colluding servers. Even if the two servers are colluding, only PIR breaks and we achieve a similar privacy factor as GPC. Additionally, we need one more round trip to transmit some randomness that the client needs to create the query.

In our protocol the workflow is as follows: The server first calls CreateDatabase to initialize the database that is already prepared for the PIR scheme. A client sends a request to the server that contains no information necessary for the protocol.[3] After the server receives the request the server sends a seed S_i necessary to call the CreateRequest algorithm. The CreateRequest function sends the queries $q_0, \dots, q_{n-1}$ to the respective server i. Each server calls the CreateResponse method that computes the query-specific answer ans sends it to the client. As last step the client calls the Verdict function that terminates the PIR and PSI sub-protocols to retrieve the potential leaked credentials of the client. It is also suitable to use our compressible PIR optimization described in Sect. 3.4 by using the 256^z partitions as blocks for our database. Since hashed values are generally distributed equally, the size of the blocks are very similar. In our case, the entries of the database are sorted, s.t. we can compress the single blocks by storing the differences of adjacent entries. This technique was also used in [TLZ$^+$17] and is an effective compression while the overhead of decompression is small. We give more details about the compression function in Sect. 3.4. An overview of the online protocol is given in Prot. 4.2 and the four algorithms are described in the following paragraphs.

CreateDatabase. In our protocol the CreateDatabase method generates the databases for all of the $n > 1$ servers. The database DB is exactly created as in Google's CreateDatabase method [TPY$^+$19] described in Sect. 4.2, i.e., DB contains $B = 256^z$ blocks. It is possible to increase z without loosing anonymity other than in Google's protocol where we should keep z as small as possible. Increasing z means that the DB contains more blocks of smaller sizes, so the communication is affected, since the last message of the server is one block and the client sends one bit for each block. In Sect. 3.4, we showed that we achieve the best communication overhead with a blocksize of $\hat{b} = \sqrt{|D|/n}$, where $|D|$ is the amount of data. With $|D| = B\hat{b}$ and $B = 256^z$, one can show that $z = \log_{256}(\hat{b}n)$ is the parameter with the best communication.

With the blocksize $\hat{b}$ and prefix parameter z, we compress each block with our compression function ρ. Afterwards, we call the $Create$ function from our query-dependent preprocessing

[3]Depending on the implementation the query could contain some metadata.

RAID-PIR scheme to retrieve the precomputed memory M_i. In parallel, the $Preprocess$ algorithm runs to fill the queue Q_i with (seed, value) pairs as described in Sect. 3.3.

After each connection setup from the client, the server pops a (seed, value) pair (S_i, A_i) from the queue Q_i and sends the seed S_i to the client form which it generates the request.

CreateRequest. In this method the client computes the two values a, H^a, idx as in the CreateRequest method of Google [TPY$^+$19] described in Sect. 4.2. The client calls the $Request$ function from the underlying PIR scheme on input idx and $S_0, \dots, S_{n-1}$ to retrieve the queries $q_0, \dots, q_{n-1}$. Server i receives the query q_i and one of the n servers - let us say server 0 - additionally receives H^a.

CreateResponse. Server 0 blinds the value H^a with its secret key b and sends the result H^{ab} to the client. Each server i computes the result P_i calling the $Response$ method of the PIR scheme on inputs q_i, DB_i, M_i and A_i and sends P_i to the client.

Verdict. The client unblinds the received H^{ab} to H^b as in Google's Verdict algorithm [TPY$^+$19]. The compressed partition $\widetilde{DB}_{idx}$ can be obtained by calling the $Combine$ method of the PIR scheme on inputs $P_0, \dots, P_{n-1}$. This partition can be decompressed to DB_{idx} using the inverted compression function ρ^{-1}. As last step the client checks if H^b is included in DB_{idx}.

We measure the PIR part of this protocol, where the communication is 1.8 MB data and the runtime is 13 seconds for a database with two billion entries.

Server		**Client**
input: username||password data D		
$b \leftarrow_{\$} \mathbb{G}$		
$(DB_i, M_i, Q_i) \leftarrow \text{CreateDatabase}(D, b)$		
		input: username u, password p
$(S_i, A_i) \leftarrow Q_i.pop()$		
	$\xrightarrow{S_i}$	
		$\text{CreateRequest}(u, p, S_i)$:
		$a \leftarrow_{\$} \mathbb{G}$
		$H \leftarrow \mathcal{H}(u\|\|p)$
		$H^a \leftarrow \text{Blind}(H, a)$
		$idx \leftarrow H[0; z]$
		$(q_0, \ldots, q_{n-1}) \leftarrow \text{PIR.Request}(idx, S_0, \ldots, S_{n-1})$
	$\xleftarrow{H^a, q_i}$	
$\text{CreateResponse}(H^a, b, q_i, DB_i, M_i, A_i)$:		
$H^{ab} \leftarrow \text{Blind}(H^a, b)$		
$P_i \leftarrow \text{PIR.Response}(q_i, DB_i, M_i, A_i)$		
	$\xrightarrow{H^{ab}, P_i}$	
		$\text{Verdict}(H^{ab}, a, P_0, \ldots, P_{n-1})$:
		$H^b \leftarrow \text{Unblind}(H^{ab}, a)$
		$\widetilde{DB}_{idx} \leftarrow \text{PIR.Combine}(P_0, \ldots, P_{n-1})$
		$DB_{ixd} \leftarrow \rho^{-1}(\widetilde{DB}_{idx})$
		return $H^b \in DB_{idx}$ // **true** iff $u\|\|p$ **in** D

Protocol 4.2: Our C3 protocol with PIR.

5 Implementation

Demmler et al. [DHS14; DHS17] published a Python implementation of their RAID-PIR scheme[1] that is based on the upPIR [Cap13] implementation of Chor et al.'s scheme [CGKS95]. They implement the crucial XOR operations in C using SSE2 intrinsics which allows XOR operations on 16 bytes in one instruction.

We develop a new C++ implementation of RAID-PIR including our query-dependent preprocessing optimizations described in Sect. 3.3. To speed up the XOR operations even more, we automatically include the currently best available Single Input Multiple Data (SIMD) instructions (we use Intel AVX for our tests) through compiler optimizations. We verified that the necessary Assembler instructions for the available SIMD unit are generated. Additionally, we develop a multi-threaded solution for the XOR operations using OpenMP.

5.1 New QDP-RAID-PIR Framework

Our new QDP-RAID-PIR framework consists of three components: the client, the server and the database generation. The components are merged in a library that contains everything necessary to run RAID-PIR. We give an example instantiation for each component that only has to be slightly adapted to the user's application.

Every component can be configured with different console outputs and parameters, e.g., with different blocksizes or redundancy parameters. For performance reasons, we define the main parameters in the preprocessor, i.e., on the one hand, all components have to be rebuilt when modifying these parameters, on the other hand, the compiler optimizations are more effective. The configurations are global for all components and located in the file *config.h*.

We test our framework on Linux Debian 10 using the GNU Compiler Collection and *CMake* for building the framework. Our framework requires at least the C++17 standard as well as the *Boost* [98] and *GMP* [Fou91] library.

We explain the specifications of the three components and give the available configuration options next.

[1]Available at `https://github.com/encryptogroup/RAID-PIR`.

Client. The client part of the library contains functions that run the RAID-PIR protocol on the client side. The two main functions are *addServer* and *request*. *AddServer* takes the IP4 address and the port of a server and connects to that server via the Transport Control Protocol (TCP). A secure channel via Transport Layer Security (TLS) can be achieved by tunneling in Linux. At this point, the server is waiting for the client to request the seed.

When the necessary servers are added the client can call *request* on input x to retrieve the block with index x. The client then starts a thread for each server to maintain the communication. Firstly, the client receives the seeds from the servers. When the client receives one seed, the client directly begins to expand it, generates the corresponding sub-queries and XORs them into the full query (see lines 5 and 8 in Listing 3.2.1c on page 12). The queries are sent to the servers again in separate threads and the client waits to receiving the answers. At the end, all answers are combined to retrieve the desired block. The block is directly returned from the Request method without any processing since this highly depends on the underlying application and is not part of the PIR protocol in general. We show an example usage of our client library in Listing 5.1.

```
#include "QDP_RAID_PIR/QDP_RAID_pirClient.h"
#include "../QDP_RAID_PIR/helper.h"

int main () {
  auto client = RAID_pirClient ();
  client.addServer ("84.121.34.21", 7765);
  client.addServer ("73.17.127.3,", 7766);
  auto block = client.request (3);
  printBytes (block.begin(), BLOCKSIZE);
}
```

Listing 5.1: Example usage of the Client component of the library. The clients connects to two servers on ports 7765 and 7766 and requests the block with index 3. Afterwards, the bytes of the block are printed to the console.

Server. The server component contains all functions for instantiating RAID-PIR as server. Each server has an ID that identifies the part of the DB it stores and processes during Private Information Retrieval (PIR) protocols (see Fig. 3.1 on page 13 for an example). The constructor for the server takes as input this ID as well as the source file of the database and the port number. We directly read the precomputed database generated with the Method of four Russians [ADKF70] as described in Sect. 3.2, i.e., the precomputation is already finished and not every server has to compute it. Since the current trend are in-memory systems, we also assume that the database can be completely stored in the RAM. When the constructor for the server is called, it also starts a process for generating the query-dependent preprocessing data (see Sect. 3.3). This thread pauses when the assigned memory for the precomputed data queue is full or under other conditions, e.g., when the flood of XOR operations is processed in order to compute the answer for a client faster. The size of the query-dependent preprocessing data queue and the events for pausing the thread can be configured as described later in this section. In every case, the thread is started after popping a (seed, value)-pair from the queue and continued when the end of an event is reached. However, if a seed is chosen and the corresponding value is already precomputed, we let the thread write it into the queue even if an event for pausing the thread occurs. This is implemented with a mutex that is locked when the computation is done and unlocked when the thread finishes the writing process.

When the *connect* function is called, the server starts listening on the previous specified port. The server can handle multiple connections in parallel in which each connection is maintained in a separate thread. We use the *Boost.Asio* library for our network implementation. A minimal example for setting up the server is given in Listing 5.2.

```
#include "QDP_RAID_PIR/QDP_RAID_pirServer.h"

int main () {
  QDP_RAID_pirServer server (0, "raw.db_preprocess.db", 7766);
  server.connect();
}
```

Listing 5.2: Example usage of the Server component of the library. The server with ID 0 listens on port 7766 and provides PIR services for the database stored in the file *raw.db_preprocess.db*.

Generate Database. The database generation component of the library contains function for creating databases and transforming databases into a structure that the server implementation can handle. Our *generateRandomDB* function takes as input a filename and writes a random generated database into a file. The specifications like how many entries the database has and how large an entry is can be specified in the *config.h* file (details follow later in this section). One can also decide if the entries are sorted before they are written into the file using Quicksort [Hoa62].

A database file starts with some meta information, namely the number of entries, the entrysize, the blocksize and the groupsize. The preprocessor definitions have to match these parameters when the server uses a given database file since the PIR protocol would not work correctly otherwise. That is why we check these values before a server proceeds to provide its service.

The function that does the preprocessing of a database is also part of the database generation component. It takes as input a database as well as a filename, performs the precomputations on the given database using the method of four Russians (Sect. 2.3), and writes it into the file. We show an example usage of this component in Listing 5.3.

Parameter	Explanation
TIME	Define to measure the times needed for single steps.
STATE	Define to print information about the current state of the PIR protocol.
DEBUGINFORMATION	Define to print debug information.
SORTING	Define to sort the database.
PAUSECONDITION	Condition for pausing the preprocessing thread. ALWAYS - pause when processing fast XOR operations for a query NEVER - never pause HALF - pause only if the queue is at least half full
SERVERS	Specifies the number of servers running RAID-PIR.
ENTRIES	Specifies the number of entries that the database has.
ENTRYSIZE	Specifies the size in bytes for each entry in the database.
BLOCKSIZE	Specifies the size in bytes for a block.
GROUPSIZE	Specifies the number of blocks that build a group for preprocessing.
CHUNKS	Specifies the redundancy parameter, i.e., the number of chunks each server has to process.
SECURITYLEVEL	Specifies the security level, i.e., the number of bits for the seeds. Default: 128.
NUMBERSEEDS	Specifies the size of the query-dependent preprocessing queue.

Table 5.1: Parameters and configuration.

```
#include "QDP_RAID_PIR/createDatabase.h"
#include "QDP_RAID_PIR/manageDatabase.h"

int main () {
  generateRandomDB("raw.db");
  auto db = readDBFromFile("raw.db");
  preprocess(db, "raw.db_preprocess.db");
}
```

Listing 5.3: Example usage of the Generate Database component of the library. We generate a random database and write it into the file *raw.db*. Afterwards, we read the database from the same file and feed it into the *preprocess* function that writes the precomputed database into the file *raw.db_preprocess.db*.

Parameters and Configurations. Our framework has a couple of configurations that can be made in the *config.h* file. We give all parameters and configurations in Tab. 5.1.

The *PAUSECONDITION* is a design decision that highly influences the performance. It specifies when the process that precomputes (seed, value)-pairs for our query-dependent preprocessing model (see Sect. 3.3) gets paused. We measure the robustness of our implementation for all three conditions in Sect. 6.2.3. The first option (ALWAYS) is to pause it when a new query from a client is received. In this case, the system uses its complete computation power to compute the answer for the client. However, if there are many queries, the queue of precomputed (seed, value)-pairs can become empty s.t. a client has to wait longer.

To avoid the situation of an empty queue, we added the option (NEVER) to never pause the preprocessing process (except when the queue is full). In this case, the server's computation power is shared between the preprocessing process and the answer computation for the client. The client has to wait longer for the server's answer than in the first option but it is less likely that the server has an empty queue.

To combine the advantage of both options, we introduce option (HALF) that only pauses the preprocessing process when the queue is less than half full. If there are less than a half entries in the queue, the preprocessing process is not paused to avoid the situation that the queue becomes completely empty. Only if the queue is at least half full, the process gets not paused and the client receives its answer more quickly. This option can also be easily extended to other thresholds than 1/2.

5.2 Fast XOR Operations

In this section, we describe how we accelerate the XOR operations for our query-dependent preprocessing RAID-PIR scheme described in Sect. 3.3. Our C++ code for the XOR operations is shown in Listing 5.4.

The C++ code shown in Listing 5.4 works as follows: A process calls the function *fastxor* to process a query by the client. It takes as input the variable *dest* where the result is stored, the variable *data* that contains the list of precomputed values described in Sect. 3.2.2, the variable *query* which was sent by the client, and the variable *groups* that specifies the number of groups that has to be processed, namely the number of groups in the flip chunk of the server. Since we implement RAID-PIR [DHS14; DHS17] in our query-dependent preprocessing model (see Sect. 3.3), the variable *dest* already contains the value from the (seed, value)-pair from the query Q_i of server i (as depicted in Prot. 3.2 on page 17). The *offset* variable in line 11 of Listing 5.4 specifies the index in *data* where the next group starts. Since we iterate over all groups in line 13 of Listing 5.4, the size of the group - namely $b \cdot 2^t$ for blocksize b and groupsize t - is added after each iteration. In line 14 of Listing 5.4, we determine the concrete index of the precomputed block to XOR. We set the groupsize $t = 8$ in our implementation since this is a good trade off between efficiency and storage capacity. Thus, our byte array *query* can be read byte by byte in each iteration. Finally, we call the inline function *xorFullBlocks* that XORs the determined block into the *dest* variable in lines 5 and 6 in Listing 5.4.

The code is compiled with the GNU Compiler Collection using the *-O3* flag that turns on most of the optimizations the compiler offers. These optimizations also cover SIMD instructions

Listing 5.4: C++ code of our XOR operations. The function xorFullBlocks takes two byte arrays as inputs, computes XOR operations over the blocksize and stores the result in the dest variable. The fastxor function takes as inputs three byte arrays and a number. It computes an XOR operation for each group over the blocksize using the xorFullBlocks function. The precomputed values for the groups are stored in the data array while the query array specifies the blocks to XOR. BLOCKSIZE and GROUPSIZEP2 are constant values defined in the configuration file (Note, that GROUPSIZEP2 is not included in Tab. 5.1 since this is a constant that a user should not modify. Concretely, it computes $2^{\text{GROUPSIZE}}$.).

```
 1 constexpr std::size_t preOffset = BLOCKSIZE * GROUPSIZEP2;
 2 
 3 inline constexpr void xorFullBlocks (std::byte *dest, const std::byte *data) {
 4 #pragma omp simd reduction(^:dest[:BLOCKSIZE])
 5   for (std::size_t i = 0; i < BLOCKSIZE; ++i) {
 6     dest[i] = data[i] ^ dest[i];
 7   }
 8 }
 9 
10 inline constexpr void fastxor (std::byte *dest, const std::byte *data, const std::byte *query, const uint64_t groups) {
11   std::size_t offset = 0;
12 #pragma omp parallel for shared (offset)
13   for (std::size_t i = 0; i < groups; ++i, offset += preOffset) {
14     std::size_t temp = offset + (static_cast<std::size_t> (query[i]) * BLOCKSIZE);
15 #pragma omp critical
16     xorFullBlocks (dest, &(data)[temp]);
17   }
18 }
```

like SSE or Intel AVX. We test our framework on machines that support Intel AVX2 or Intel AVX-512 allowing to process (XOR) operations on 256 or 512 bits, respectively. Especially, line 5 of Listing 5.4 is optimized s.t. it is processed with the Intel AVX instruction set.

In the not yet discussed line 12 in Listing 5.4 we indicate the parallelization of our *fastxor* algorithm. By doing this, we use *OpenMP* (Open Multi-Processing) that allows shared memory programming on systems having multiple processors and / or processor cores. OpenMP is specialized for parallelizing loops that perfectly fits in our application.

OpenMP distributes the work that has to be done in the for loop shown in lines 4, 12, and 15 in Listing 5.4 over multiple threads depending on the underlying system. In line 12, we define the *offset* variable as shared variable over all the threads, i.e., it is well-defined in each thread. The *critical* keyword in line 15 forces that line 16 is executed by only one thread in parallel since the *dest* variable would get overwritten in the *xorFullBlocks* function. In line 4 the SIMD instructions à la Intel AVX are included in the loop. To avoid writing conflicts when multiple threads access the *dest* variable, we use the *reduction* keyword in line 4 that ensures that XOR operations on the *dest* variable are only allowed by one thread in parallel.

Pipeline. As transmitting queries takes some time, we implement a pipelining approach, where a server does not wait until the query is completely transmitted, rather directly starts to compute the answer when enough data is available (each byte of the query can be directly processed without knowledge of the remaining query). So, we move a part of the XOR operations to the communication phase and continue with the remaining groups with the code shown in Listing 5.4. As this pipelining approach is quite complex, we do not include the SIMD and parallelization optimizations.

We realize this pipelining technique with an asynchronous receiving operation, where the data is written into a buffer and the program does not wait until the receiving process terminates (as synchronous receiving would do). In a separate thread, we continuously observe if enough data for the next XOR operations is already written into the buffer and process the part of the query when this is the case. When the receiving process terminates, we interrupt our pipeline and run the fastxor function from Listing 5.4 for the not yet processed sub-query.

6 Evaluation

In this chapter, we evaluate our implementation from Chapt. 5. We define our system setup in Sect. 6.1. In Sect. 6.2.1 we evaluate the query-dependent preprocessing RAID-PIR scheme described in Sect. 3.3.2. At last, we evaluate the Private Information Retrieval (PIR) part of our Compromised Credential Checking (C3) protocol (see Sect. 4.3) in Sect. 6.3 to show practicability of C3 in combination with PIR.

6.1 Setup

In this section, we give details about the machines we use for performing our benchmarks. For the RAID-PIR benchmarks in Sect. 6.2 we use a client and servers that are running Arch Linux and have a 16 Core Intel Core i9-7960X processor and 128 GB DDR4 RAM. The processor allows to use up to AVX-512 instructions that we use for our RAID-PIR benchmarks in Sect. 6.2.

For a more realistic C3 scenario we deploy two servers as x1e.32xlarge instances on Amazon EC2 located in Europe and Virginia. The x1e.32xlarge instance is powered by four Intel Xeon E7 8880 v3 processors and has 3904 GB RAM that suffices to load our whole database. The processors support AVX2 that we use for our C3 benchmarks in Sect. 6.3.

6.2 RAID-PIR Benchmarks

In this section, we benchmark our query-dependent preprocessing RAID-PIR scheme from Sect. 3.3.2. In Sect. 6.2.1 we run experiments for generating the database. Thereafter, we compare the online time and communication of our implementation in Sect. 6.2.2. Finally, we test the robustness in Sect. 6.2.3 and conclude that our implementation can only handle floods for a small time period.

6.2.1 Generating the Database

In this section, we evaluate the database generation. This includes the generation of the database itself, the preprocessing to map the database into a larger preprocessing database, and a comparison between the sizes of the two databases.

Number of Entries N	Generate Random Bits (ms)	Quicksort (ms)
100 000	584	956
1 000 000	5 271	9 535
10 000 000	52 288	105 973
50 000 000	260 708	535 622

Table 6.1: Times in ms needed for generating a random database and sorting it using the Quicksort algorithm. The database consists N entries of size 32 bytes each. We note that the blocksize b is independent from the database generation and thus has not to be defined yet.

Number of Entries N	Preprocessing (ms)		
	$b = 1000$ **bytes**	$b = 5000$ **bytes**	$b = 10000$ **bytes**
100 000	405	386	383
1 000 000	3 383	3 356	3 358
10 000 000	33 732	33 505	33 610
50 000 000	171 838	172 316	170 991

Table 6.2: Times in ms for preprocessing using the method of four Russians [ADKF70] with varying blocksizes b bytes.

Database Generation Tab. 6.1. Firstly, we evaluate the times needed for generating the databases without any preprocessing. We generate four databases that consists of N entries where each entry represents a hash value consisting of 32 bytes. For our C3 application, we need to sort the database and thus we also show the times needed for sorting separately. The total running time is the sum of the times needed for generating random bits and sorting them with our Quicksort implementation. The results are given in Tab. 6.1. We see a linear dependence on the number of entries and the time needed for generating random bits and sorting.

Preprocessing Tab. 6.2. We now measure the times for the preprocessing using the method of four Russians [ADKF70] as described in Sect. 3.2.2. For our further experiments, we choose a constant groupsize of $t = 8$ and different block sizes b that also influences the precomputed memory P_i of server i. Our results are depicted in Tab. 6.2. We see that the preprocessing time is independent from the blocksize b, what we expected, since the amount of work is independent of the blocksize - only the number of groups differ. As for the database generation, we see a linear dependency on the number of entries and the time.

Database sizes Tab. 6.3. At last, we look at the sizes of the databases and the corresponding precomputed databases. The results are shown in Tab. 6.3. It is obvious, that the size of the databases are again linear in the number of database entries. The size of the precomputed

Number of Entries N	Database Size (MB)	Preprocessing Database Size (MB)
100 000	3.2	102.4
1 000 000	32.0	1 024.0
10 000 000	320.0	10 240.0
50 000 000	1 600.0	51 200.0

Table 6.3: Sizes of the randomly generated database and the corresponding preprocessing database for various number of database entries with blocksize $b = 1\,000$ bytes and groupsize $t = 8$.

database is independent of the blocksize, what can be easily shown as follows: Assume a database of size $|D|$ with blocksize b and a groupsize of t. We have to assign the $B = |D|/b$ blocks to groups of size t, i.e., we have $|D|/(bt)$ groups. For each group we have to store 2^t values of b bytes, s.t. we have to store $|D|/(bt) \cdot 2^t \cdot b = 2^t \cdot |D|/t$ bytes, which is independent of the blocksize b. We also see that the database size in our implementation exactly matches with the theoretical $2^t \cdot |D|/t$ bytes.

6.2.2 Query and Response

Here, we look at the online time of our RAID-PIR protocol. This includes sending the seeds to the client, receiving the queries from the client, computing the answers, sending these to the client, and finally combining them to the desired block. We run the experiments on a LAN and a WAN setting with $n = 2$ and $n - 3$ servers.

The majority of the execution time is spent on the huge number of XOR operations. In Sect. 3.5, we showed that each server has to process kb/t XOR operations. By setting $k = |D|/n$ and $|D| = B/b$, we achieve a complexity of $|D|/(nt)$ XOR operations and thus the number of XOR operations is independent of the blocksize b. Only the communication depends on the blocksize b. We showed in Sect. 3.4 that our RAID-PIR protocol has the best communication complexity $\mathcal{C}(b)$ at a blocksize of $b = \sqrt{|D|/n}$ with $|D| = 32N$ when each entry in the database is a 32 byte hash value. So, we test for each database with three different blocksizes, one that theoretically leads to the best communication, a smaller and a bigger one.

We summarize the upload and download times in our measurements. As described in Sect. 3.3.1, the client uploads a $\kappa = 128$ bit seed and the queries, while the download only consists of one block of size b for each server.

We wait until the queue for (seed, value)-pairs is full. For the experiments with 3 running servers we choose the redundancy parameter $r = 2$, i.e., each server has to process two chunks - one online and one offline.

We give the online computation time for our implementation of the RAID-PIR scheme [DHS14; DHS17] and the query-dependent preprocessing (QDP) RAID-PIR scheme. We do not provide a correct implementation of the original RAID-PIR scheme, rather adapt the query size and

Entries N	Blocksize b (bytes)	Comm. (ms)	Computation (ms)		Up (kB)	Down (kB)
			RAID-PIR	QDP RAID-PIR		
100 000	800	2	13	8	4.00	1.66
	1 265	2	13	9	2.53	2.59
	2 000	3	13	8	1.60	4.06
1 000 000	1 000	6	38	24	32.00	2.06
	4 000	3	40	23	8.00	8.06
	10 000	7	42	27	3.20	20.06
10 000 000	8 000	11	276	173	40.00	16.06
	12 650	6	273	174	25.30	25.36
	20 000	9	273	177	16.00	40.06
50 000 000	10 000	16	853	512	160.00	20.06
	28 285	14	846	517	56.57	56.63
	40 000	16	852	524	40.00	80.06

Table 6.4: Online phase LAN benchmarks for $n = 2$ servers and redundancy parameter $r = 2$.

the necessary number of XOR operations since this takes the main runtime. For a realistic comparison, we do not need to implement the correct scheme. Note, that we only measure the communication time for the QDP RAID-PIR scheme since the amount of data is equal for both schemes and thus we expect similar results.

The benchmarks for the 1Gbit LAN setting are depicted in Tab. 6.4 for $n = 2$ servers and in Tab. 6.5 for $n = 3$ servers. The benchmarks for the WAN setting are given in Tab. 6.6 for $n = 2$ servers and in Tab. 6.7 for $n = 3$ servers. We see that the computation time does not grow linearly although the number of XOR operations grows linearly. The reason is the synchronization overhead of the parallelism. OpenMP needs some time to synchronize the threads and thus the parallel execution is more effective for larger databases.

We can also see that the computation time for the larger databases in the WAN setting is substantially smaller than for the LAN setting with the same inputs. The reason for this is that we use a pipelining approach where the servers do not wait until the query is completely transmitted rather directly start the computation of the answer when enough data is available, i.e., some computation time moves to the communication time as described in Sect. 5.2. So, we see that the total runtime (communication and computation) for the WAN setting is only slightly worse.

We see a significant improvement of the computation time of our QDP RAID-PIR scheme compared to the original RAID-PIR scheme [DHS14; DHS17]. For $n = 2$ servers, we achieve improvements up to 40% (see, e.g., the three rows for $N = 50$ million database entries in Tab. 6.4), and for $n = 3$ servers, the improvement is up to 66% (see, e.g., the last row of Tab. 6.5).

The amount of upload and download data is measured and matches to the theoretical formulas given in Sect. 3.5. We see that we have the best communication by choosing the optimal

Entries N	**Blocksize** b **(bytes)**	**Comm. (ms)**	**Computation (ms)**		**Up (kB)**	**Down (kB)**
			RAID-PIR	**QDP RAID-PIR**		
100 000	800	2	6	2	4.00	2.50
	1 033	2	6	3	3.10	3.16
	2 000	2	6	2	1.60	6.10
1 000 000	1 000	5	69	24	32.00	3.10
	3 265	4	72	26	9.80	9.89
	10 000	5	71	21	3.20	30.10
10 000 000	8 000	11	376	127	40.00	24.10
	10 372	8	381	132	30.85	31.21
	20 000	10	385	128	16.00	60.10
50 000 000	10 000	16	1 303	438	160.00	30.10
	23 095	14	1 287	425	69.28	69.38
	40 000	17	1 308	431	40.00	120.10

Table 6.5: Online phase LAN benchmarks for $n = 3$ servers and redundancy parameter $r = 2$.

Entries N	**Blocksize** b **(bytes)**	**Comm. (ms)**	**Computation (ms)**		**Up (kB)**	**Down (kB)**
			RAID-PIR	**QDP RAID-PIR**		
100 000	800	15	12	9	4.00	1.66
	1 265	13	12	8	2.53	2.59
	2 000	14	11	11	1.60	4.06
1 000 000	1 000	28	61	38	32.00	2.06
	4 000	26	64	34	8.00	8.06
	10 000	29	63	36	3.20	20.06
10 000 000	8 000	99	185	90	40.00	16.06
	12 650	89	182	89	25.30	25.36
	20 000	97	185	90	16.00	40.06
50 000 000	10 000	197	651	412	160.00	20.06
	28 285	189	638	389	56.57	56.63
	40 000	196	644	403	40.00	80.06

Table 6.6: Online phase WAN benchmarks for $n = 2$ servers and redundancy parameter $r = 2$.

Entries N	Blocksize b (bytes)	Comm. (ms)	Computation (ms)		Up (kB)	Down (kB)
			RAID-PIR	QDP RAID-PIR		
100 000	800	8	16	6	4.00	2.50
	1 033	7	16	6	3.10	3.16
	2 000	8	16	6	1.60	6.10
1 000 000	1 000	21	58	23	32.00	3.10
	3 265	19	56	19	9.80	9.89
	10 000	22	58	19	3.20	30.10
10 000 000	8 000	64	324	83	40.00	24.10
	10 372	61	313	82	30.85	31.21
	20 000	63	318	86	16.00	60.10
50 000 000	10 000	201	1 170	293	160.00	30.10
	23 095	182	1 146	283	69.28	69.38
	40 000	202	1 186	287	40.00	120.10

Table 6.7: Online phase WAN benchmarks for $n = 3$ servers and redundancy parameter $r = 2$.

blocksize $\sqrt{32N/n}$, where the amount of upload and download data is almost equal. In every case, the communication time is the best for the optimal blocksize, while the computation time is not much affected.

6.2.3 Robustness

In this section, we look at the robustness of our scheme, i.e., how many queries our implementation can reliably handle. Therefore, we let three clients permanently send requests every 500 ms (i.e., six requests in a second) and observe how many requests are currently unanswered over five minutes. For this benchmark, we only test the WAN setting as this one is more realistic in a real world deployment. Furthermore, we choose the optimal blocksize $b = \sqrt{32N/n}$ for a given number of database entries N. We choose $n = 2$ PIR server and test all three pause conditions as described in Sect. 5.1. For this setup, we assume a queue capacity of 500 (seed, value)-pairs and start the experiments with a full queue. The results are shown in Fig. 6.1.

We observe that our implementation is not robust against a flood of requests for a long time. Let ALWAYS, NEVER and HALF are the pause conditions as shown in Tab. 5.1 on page 37. The pause conditions ALWAYS and HALF for the smaller database can handle the flood reliably for 1.5 minutes and 0.75 minutes, respectively. However, the ALWAYS pause condition becomes very inefficient after the (seed, value)-pair queue is empty. The NEVER pause condition has a constant rate but cannot handle high floods even over a short time period. In most scenarios, the HALF pause condition is the best choice since it allows high floods for a short time period while it still has the same rate as the NEVER pause condition afterwards.

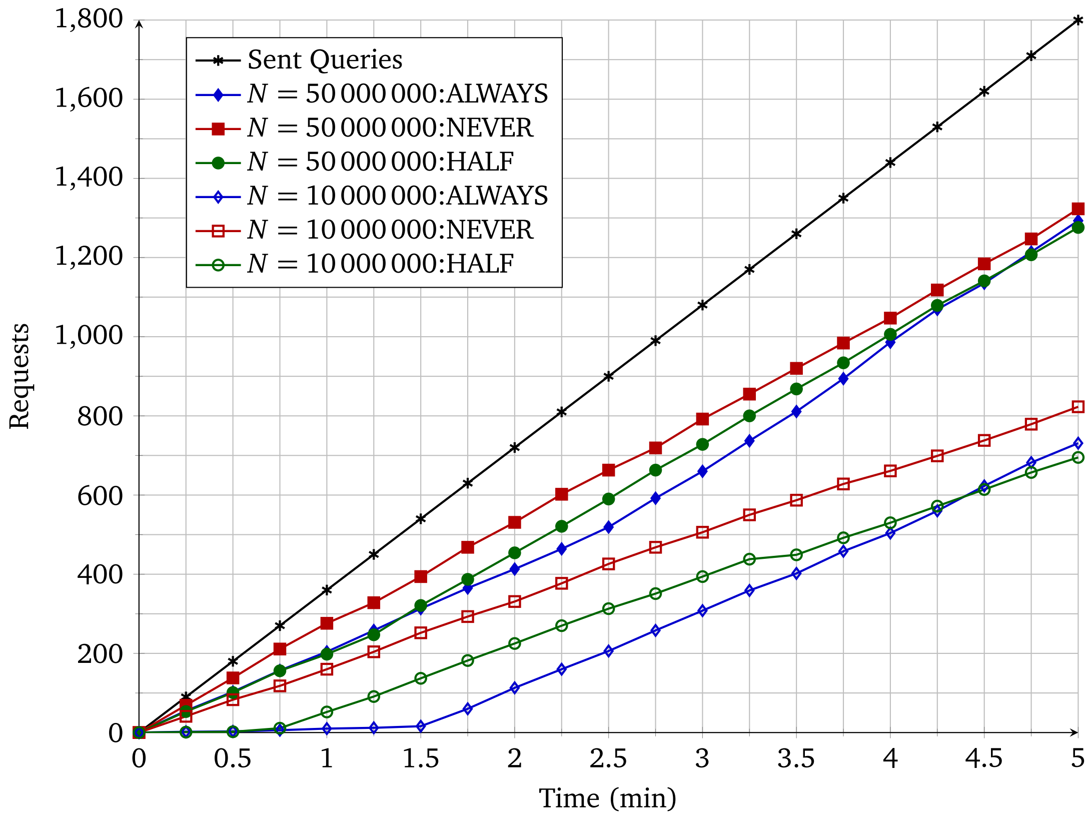


Figure 6.1: Robustness benchmarks: the black line gives the number of so far sent queries while the other lines give the number of queries that are not yet processed, i.e., the lower the lines the better the performance. ALWAYS (blue) describes the configuration to pause the preprocessing process when an answer is computed, NEVER (red) never pauses the process and HALF (green) only pauses the process if the (seed, value)-pair queue is less than half full. The experiments are measured for databases with $N = 10\,000\,000$ (unfilled markers) and $N = 50\,000\,000$ (filled markers) entries. We start each experiment with a full queue of 500 (seed, value)-pairs.

6.3 Benchmarks for the PIR Part of our C3 Protocol

In this section, we deploy the PIR part of our C3 protocol described in Sect. 4.3 on $n = 2$ Amazon EC2 server instances to simulate a real-world scenario. As client, we choose a laptop with 32 GB DDR4 memory and an 8-Core Intel i9 CPU with 2,4 GHz. For this use case, we generate a random database with two billion entries, that almost covers the data breaches Collection 1 - 5 [Gre19]. Since hash values are distributed uniformly at random and thus their blinded value also does (otherwise the hash function would not be collision-resistant), a random database simulates a real-world data breach perfectly. To shorten our database, we compress the database with the compression function used in [TLP+17] and explained in Sect. 4.3.

For a database with $N = 2\,000\,000\,000$ entries, we have an optimal blocksize of $\hat{b} = \sqrt{32N/2} \approx 178\,885$ bytes. As shown in Sect. 4.3, the optimal prefix parameter is $z = \log_{256}(\hat{b}n) \approx 2.3$. Since we achieve a better compression for a small prefix parameter, we choose $z = 2$, s.t. each block has the same 2 byte prefix and thus, we would leak the same information as Google Password Checkup (GPC) [TPY+19] if our PIR protocol is broken, e.g., if the servers collude.

In our randomly generated database, the maximum number of entries with the same prefix is 31 152, which yields roughly 40 million dummy entries in the database, since we have to fill each block, that has less than 31 152 entries, with dummies. Without any compression, we achieve the blocksize $b = 32 \cdot 31\,152 = 996\,000$ bytes. After our compression, we get a blocksize of $b = 880\,837$ bytes, which results in a database of 57.7 GB, while the uncompressed database was $65,3$ GB, so we achieve a compression by $\approx 12\%$.

Runtime and Communication. We run the PIR part of our C3 protocol on two x1e.32xlarge instances, both located in Frankfurt. The client is located in Darmstadt, roughly 27 km direct distance (air-line distance) away. The connections between the client and the servers has a latency of 10 ms. Server 0 and Server 1 need 118 minutes and 135 minutes to generate the database including preprocessing, respectively. Both servers need around $14,6$ minutes to compute 100 (seed, value)-pairs for the query-dependent preprocessing queue. We start the experiments with a full queue.

We measure the online phase by letting a client retrieve a single block and repeat this ten times. On average, the client waits ≈ 13 seconds until it receives the desired block, from which ≈ 9 seconds are spent on the XOR operations. So, the client has to wait approximately 4.5 seconds longer than in Thomas et al's GPC protocol [TPY+19]. However, the measurements from our protocol only includes the PIR part. For our whole C3 protocol, the client has to additionally run a Private Set Intersection (PSI). Fortunately, the server's PSI set consists of only 31 152 entries, so we expect a small overhead for the PSI part.

The total communication is 1,8 MB of which 1,7 MB is the size of the downloaded block. Note that for an optimal blocksize $\hat{b}$, the upload and download is almost equal. However, since our C3 protocol requires that the blockindex is equal to the prefix of each entries in a block, we cannot choose any blocksize. Another obstacle is the compression, which makes it hard to predict a blocksize.

7 Conclusion

Summary. Compromised Credential Checking (C3) protocols like HaveIBeen-Pwned (HIBP) [Hun19] and Google Password Checkup (GPC) [TPY$^+$19] allow users to check if their credentials are leaked in a data breach. However, all of these protocols leak a prefix of the hashed credentials, which is enough information to exploit a credential stuffing attack [LPA$^+$19]. In this thesis, we circumvent the prefix leakage problem by providing a C3 protocol that is based on multi-server Private Information Retrieval (PIR) [CGKS95]. Since PIR hides the index of the data block to retrieve, there is no information revealed to the servers. So, we developed the first C3 protocol that fulfills perfect user anonymity and has a two server PIR runtime of roughly 12 seconds for a database with 2 billion entries, which is nearly the size of the well-known data breaches Collection 1-5 [Gre19].

We achieved this acceptable runtime by developing the new query-dependent preprocessing PIR model, that moves at least half of the online work (depending on the number of servers) to an offline phase at the cost of a half more round trip time. We implemented this feature in the RAID-PIR scheme [DHS14; DHS17] and provide our new query-dependent preprocessing RAID-PIR C++ framework. Additionally, we compressed each block of the PIR database to reduce the amount of data to store and to transmit, and generalized this approach to Compressible PIR.

Future Work. We provide an implementation of our new query-dependent preprocessing RAID-PIR scheme, but not yet an implementation of our full C3 protocol. To measure more realistic times for our whole C3 protocol, one could extend our PIR framework with the remaining steps of our C3 protocol. This includes the determination of the block index and the Private Set Intersection (PSI) protocol.

A drawback of multi-server PIR is that it requires multiple non-colluding servers that might be hard to realize in practice [TPY$^+$19; LPA$^+$19]. An interesting research direction is to develop techniques that prevent malicious servers to collude. Secret Sharing [Sha79] faces the same problem, where a secret is distributed over multiple parties and only a subset of the parties may collude. The PIR query can be considered as secret and is shared between the servers. An interesting approach can be the usage of Intel SGX, where the server computes the answer in an Intel SGX enclave and the client communicates directly with the enclave over a secure channel. Tamrakar et al. [TLP$^+$17] use a similar approach for their private membership test scheme.

Bibliography

[98] **"Boost C++ Libraries"**. `https://www.boost.org`. 1998.

[ADKF70] V. L. ARLAZAROV, E. A. DINIC, M. A. KRONROD, I. A. FARADZEV. **"On Economical Construction of the Transitive Closure of an Oriented Graph"**. In: *134. Journal of USSR Academy of Sciences* (1970), pp. 1209–1210.

[ALS14] D. AUGOT, F. LEVY-DIT-VEHEL, A. SHIKFA. **"A Storage-Efficient and Robust Private Information Retrieval Scheme Allowing Few Servers"**. In: *13. International Conference on Cryptology and Network Security (CANS'14)*. Springer, 2014, pp. 222–239.

[Bar09] G. BARD. **"Algebraic Cryptanalysis"**. Springer, 2009.

[BIM00] A. BEIMEL, Y. ISHAI, T. MALKIN. **"Reducing the Servers Computation in Private Information Retrieval: PIR with Preprocessing"**. In: *20. International Conference on Advances in Cryptology (CRYPTO'00)*. Springer, 2000, pp. 55–73.

[BM84] G. R. BLAKLEY, C. MEADOWS. **"Security of Ramp Schemes"**. In: *5. International Conference on Advances in Cryptology (CRYPTO'84)*. Springer, 1984, pp. 242–268.

[Cap13] J. CAPPOS. **"Avoiding theoretical Optimality to Efficiently and Privately Retrieve Security Updates"**. In: *17. International Conference on Financial Cryptography and Data Security (FC'13)*. Springer, 2013, pp. 386–394.

[CGKS95] B. CHOR, O. GOLDREICH, E. KUSHILEVITZ, M. SUDAN. **"Private Information Retrieval"**. In: *36. Symposium on Foundations of Computer Science FOCS'95*. IEEE, 1995, pp. 41–50.

[CMO00] G. D. CRESCENZO, T. MALKIN, R. OSTROVSKY. **"Single Database Private Information Retrieval Implies Oblivious Transfer"**. In: *19. International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT'00)*. Springer, 2000, pp. 122–138.

[DG15] Z. DVIR, S. GOPI. **"2 Server PIR with Sub-Polynomial Communication"**. In: *47. ACM Symposium on Theory of Computing (STOC'15)*. ACM, 2015, pp. 577–584.

[DHS14] D. DEMMLER, A. HERZBERG, T. SCHNEIDER. **"RAID-PIR: Practical Multi-Server PIR"**. In: *6. ACM Cloud Computing Security Workshop (CCSW'14)*. ACM, 2014, pp. 45–56.

[DHS17] D. Demmler, M. Holz, T. Schneider. **“OnionPIR: Effective Protection of Sensitive Metadata in Online Communication Networks”**. In: *15. International Conference on Applied Cryptography and Network Security (ACNS’17)*. Springer, 2017, pp. 599–619.

[DJ01] I. Damgård, M. Jurik. **“A Generalisation, a Simplification and Some Applications of Paillier’s Probabilistic Public-Key System”**. In: *4. International Workshop on Practice and Theory in Public Key Cryptography (PKC’01)*. Springer, 2001, pp. 119–136.

[ENZ16a] ENZOIC. **“Detect Compromised Passwords”**. `https://www.enzoic.com/`. 2016.

[ENZ16b] ENZOIC. **“LastPass selects PasswordPing for Compromised Credential Screening”**. `https://www.enzoic.com/lastpass-selects-passwordping-for-compromised-credential-screening/`. 2016.

[Fou91] F. S. Foundation. **“The GNU Multiple Precision Arithmetic Library”**. `https://gmplib.org`. 1991.

[GH19] C. Gentry, S. Halevi. **“Compressible FHE with Applications to PIR”**. In: *17. International Conference on Theory of Cryptography (TCC’19)*. To appear. Online: `https://eprint.iacr.org/2019/733`. Springer, 2019.

[Gol07] I. Goldberg. **“Improving the Robustness of Private Information Retrieval”**. In: *28. IEEE Symposium on Security and Privacy (S&P’07)*. IEEE, 2007, pp. 131–148.

[Gre19] A. Greenberg. **“Hackers are Passing Around a Megaleak of 2.2 Billion Records”**. `https://www.wired.com/story/collection-leak-usernames-passwords-billions/`. 2019.

[HHG13] R. Henry, Y. Huang, I. Goldberg. **“One (Block) Size Fits All: PIR and SPIR with Variable-Length Records via Multi-Block Queries”**. In: *20. Symposium on Network & Distributed System Security (NDSS’13)*. The Internet Society, 2013, pp. 364–373.

[Hoa62] C. A. R. Hoare. **“Quicksort”**. In: *The Computer Journal* (1962), pp. 10–16.

[Hun19] T. Hunt. **“Have I Been Pwnd?”** `https://haveibeenpwned.com/`. 2019.

[IR89] R. Impagliazzo, S. Rudich. **“Limits on the Provable Consequences of One-way Permutations”**. In: *21. ACM Symposium on Theory of Computing (STOC’89)*. ACM, 1989, pp. 44–61.

[KLL$^+$15] A. Kiayias, N. Leonardos, H. Lipmaa, K. Pavlyk, Q. Tang. **“Optimal Rate Private Information Retrieval from Homomorphic Encryption”**. In: *Proceedings on Privacy Enhancing Technologies (PoPETs’15)* (2015), pp. 222–243.

[KLS$^+$17] Á. Kiss, J. Liu, T. Schneider, N. Asokan, B. Pinkas. **“Private Set Intersection for Unequal Set Sizes with Mobile Applications”**. In: *Proceedings on Privacy Enhancing Technologies (PoPETs’17)* (2017), pp. 177–197.

[KO97] E. KUSHILEVITZ, R. OSTROVSKY. **"Replication is not Needed: Single Database, Computationally-Private Information Retrieval"**. In: *38. Symposium on Foundations of Computer Science (FOCS'97)*. IEEE, 1997, pp. 364–373.

[KRS$^+$19] D. KALES, C. RECHBERGER, T. SCHNEIDER, M. SENKER, C. WEINERT. **"Mobile Private Contact Discovery at Scale"**. In: *28. USENIX Security Symposium (USENIX Security'19)*. USENIX Association, 2019, pp. 1447–1464.

[Kuk19] M. KUKETZ. **"Chrome-Add-on: Password Checkup übermittelt Domainname"**. `https://www.kuketz-blog.de/chrome-add-on-password-checkup-uebermittelt-domainname/`. 2019.

[LP17] H. LIPMAA, K. PAVLYK. **"A Simpler Rate-Optimal CPIR Protocol"**. In: *21. International Conference on Financial Cryptography and Data Security (FC'17)*. Springer, 2017, pp. 621–638.

[LPA$^+$19] L. LI, B. PAL, J. ALI, N. SULLIVAN, R. CHATTERJEE, T. RISTENPART. **"Protocols for Checking Compromised Credentials"**. In: *26. ACM Conference on Computer and Communication Security (CCS'19)*. ACM, 2019, pp. 1387–1403.

[Mat18] N. MATATALL. **"New Improvements and Best Practices for Account Security and Recoverability"**. `https://github.blog/2018-07-31-new-improvements-and-best-practices-for-account-security-and-recoverability/`. 2018.

[Mea86] C. A. MEADOWS. **"A More Efficient Cryptographic Matchmaking Protocol for Use in the Absence of a Continuously Available Third Party"**. In: *7. IEEE Symposium on Security and Privacy (S&P'86)*. IEEE, 1986, pp. 134–137.

[MG08] C. A. MELCHOR, P. GABORIT. **"A fast Private Information Retrieval Protocol"**. In: *54. IEEE Symposium on Information Theory (ISIT'08)*. IEEE, 2008, pp. 1848–1852.

[OG11] F. OLUMOFIN, I. GOLDBERG. **"Revisiting the Computational Practicality of Private Information Retrieval"**. In: *15. International Conference on Financial Cryptography and Data Security (FC'11)*. Springer, 2011, pp. 158–172.

[Pai99] P. PAILLIER. **"Public-key Cryptosystems Based on Composite Degree Residuosity Classes"**. In: *17. International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT'99)*. Springer, 1999, pp. 223–238.

[Sha79] A. SHAMIR. **"How to Share a Secret"**. In: *Communications of the ACM* (1979), pp. 612–613.

[Shi19] J. SHINER. **"Finding Pwned Passwords with 1password"**. `https://blog.1password.com/finding-pwned-passwords-with-1password/`. 2019.

[TLP$^+$17] S. TAMRAKAR, J. LIU, A. PAVERD, J.-E. EKBERG, B. PINKAS, N. ASOKAN. **"The Circle Game: Scalable Private Membership Test Using Trusted Hardware"**. In: *12. ACM Asia Conference on Computer and Communications Security (ASIACCS'17)*. ACM, 2017, pp. 31–44.

[TLZ+17] K. THOMAS, F. LI, A. ZAND, J. BARRETT, J. RANIERI, L. INVERNIZZI, Y. MARKOV, O. COMANESCU, V. ERANTI, A. MOSCICKI, e. AL. **"Data Breaches, Phishing, or Malware? Understanding the Risks of Stolen Credentials"**. In: *24. ACM Conference on Computer and Communications Security (CCS'17)*. ACM, 2017, pp. 1421–1434.

[TPY+19] K. THOMAS, J. PULLMAN, K. YEO, A. RAGHUNATHAN, P. G. KELLEY, L. INVERNIZZI, B. BENKO, T. PIETRASZEK, S. PATEL, D. BONEH, E. BURSZTEIN. **"Protecting Accounts from Credential Stuffing with Password Breach Alerting"**. In: *28. USENIX Security Symposium (USENIX Security'19)*. USENIX Association, 2019, pp. 1556–1571.

[ZS14] L. F. ZHANG, R. SAFAVI-NAINI. **"Verifiable Multi-server Private Information Retrieval"**. In: *12. International Conference on Applied Cryptography and Network Security (ACNS'14)*. Springer, 2014, pp. 62–79.

List of Figures

List of Tables

List of Protocols

List of Listings

List of Abbreviations

C3	Compromised Credential Checking
DOS	Denial of Service
FSB	Frequency-Smoothing Bucketization
GPC	Google Password Checkup
HIBP	HaveIBeenPwned
IDB	ID-based Bucketization
PIR	Private Information Retrieval
PSI	Private Set Intersection
RAID	Redundant Array of Inexpensive Disks
SIMD	Single Input Multiple Data