GPU-accelerated PIR with Client-Independent Preprocessing for Large-Scale Applications

Daniel Günther
Technical University of Darmstadt

Maurice Heymann
Technical University of Darmstadt

Benny Pinkas
Bar-Ilan University

Thomas Schneider
Technical University of Darmstadt

Abstract

Multi-Server Private Information Retrieval (PIR) is a cryptographic protocol that allows a client to securely query a database entry from $n \geq 2$ servers of which less than t can collude, s.t. the servers learn no information about the query. Highly efficient PIR could be used for large-scale applications like *Compromised Credential Checking* (C3) (USENIX Security'19), which allows users to check whether their credentials have been leaked in a data breach. However, state-of-the art PIR schemes are not efficient enough for fast online responses at this scale.

In this work, we introduce *Client-Independent Preprocessing* (CIP) PIR that moves $(t-1)/n$ of the online computation to a local, client independent, preprocessing phase suitable for efficient batch precomputations. The online performance of CIP-PIR improves linearly with the number of servers n. We show that large-scale applications like C3 with PIR are practical by implementing our CIP-PIR scheme using a parallelized CPU implementation. To the best of our knowledge, this is the first multi-server PIR scheme whose preprocessing phase is completely independent of the client, and where online performance simultaneously improves with the number of servers n. In addition, we accelerate for the first time the huge amount of XOR operations in multi-server PIR with GPUs. Our GPU-based CIP-PIR achieves an improvement up to factor $2.1\times$ over our CPU-based implementation for $n = 2$ servers, and enables a client to query an entry in a 25 GB database within less than 1 second.

1 Introduction

The main motivation for this work is improving the security and privacy of large-scale applications of Private Information Retrieval (PIR) like Compromised Credential Checking (C3 – checking whether a user's credentials appear in a database of compromised credentials) [48], private blocklists in web browsers [33], and epidemiological modeling [28]. The current state of the art of C3 implementations must leak some information about each query (namely, about the user's credentials), in order to support low response latency and to scale to the sizes of existing datasets [35, 48] . This leakage allows certain attacks [35].

We present a new PIR construction with offline preprocessing, which hides all information about the queries and on top reduces the response latency compared to all existing PIR schemes. This construction works in a setting where the responses are computed by n servers of which less than t can collude. We discuss in §1.3 the applicability of this setting to C3, as well as to other applications. On top of this, we implement for the first time multi-server PIR on GPUs in order to accelerate (a) large parts of the online computations, and (b) the offline preprocessing phase of our new PIR scheme, by batching multiple queries independently of the client.

1.1 Private Information Retrieval (PIR)

Private Information Retrieval allows to securely and privately access data from a public database whereby the servers do not learn any information about the query nor the accessed data. State of the art single-server PIR protocols like SealPIR [2] are often not efficient enough for large-scale databases as their response time is already a couple of seconds for databases with only a few million entries (see also Fig. 5).

Hence, we build a new multi-server PIR protocol called CIP-PIR. As a basis, we use the RAID-PIR protocol [15, 16], which is a multi-server PIR scheme with $n \geq 2$ servers of which less than t can collude that extends Chor et al.'s PIR scheme [10]. Motivated by the very fast response times required for large-scale applications, we aim to preprocess a large part of the PIR protocol. For this, we introduce the *Client-Independent Preprocessing* (CIP) PIR model which lets the servers choose a part of the client's query in the RAID-PIR protocol and moves $(t-1)/n$ of its computation to an offline preprocessing phase which is even *independent* of the client. Now, this phase can be batch processed for all clients together resulting in faster amortized preprocessing and hence total time. We also show how to compress the PIR database to improve storage and computation in PIR which is of independent interest.

We show corresponding improvements by implementing our CIP-PIR protocol on a CPU and obtain up to factor $n\times$ better online runtime than RAID-PIR without decreasing the throughput. Moreover, while RAID-PIR trades security for better performance (only less than $t \leq n$ servers may collude), the performance of the online phase of our CIP-PIR scheme becomes *independent* of the collusion threshold t.

We further improve the runtime of our CIP-PIR protocol by massively parallel computations of the preprocessing and online computations on a GPU. The GPU-accelerated implementation highly profits from batching multiple queries in the preprocessing phase as expensive memory transfers of portions of the database are amortized.

1.2 Large-Scale PIR Applications

There is a high interest in efficient PIR for multiple applications. Our CIP-PIR construction can be used in any PIR application which requires low latency for large-scale data. An important and omnipresent application in the context of compromised credential checking (C3) is described below. Further applications with these requirements are, for example, private queries to medical and patent databases [4], anonymous messaging [16], Tor clients downloading a list of nodes, and certificate/key transparency. A recently proposed application which requires low latency is to support private queries by browsers to blocklists of malware-hosting websites, as in Google's "Safe Browsing" blocklist [33]. Another potential future application, which is motivated by the push for privacy-preserving advertising and the elimination of cross-site user tracking, is serving advertisements to users: The goal would be for the user's machine to locally decide on ads that best target the user, and then fetch these ads privately using PIR. (Of course, this future ad system will also require additional privacy-preserving mechanisms, such as for profiling users interests, and for exposure measurement and billing.)

Compromised Credential Checking (C3). Data breaches occur more and more in the recent years. These breaches contain highly sensitive information about the users, e.g., their passwords and usernames. The most prominent breach contains more than two billion credentials and is called Collection 1-5 [26]. Thomas et al. [47] showed that 6.9% of the breached credentials are still in use even on non-exposed platforms. This enables credential stuffing attacks, where an adversary compromises accounts by trying leaked passwords on other services. Usually, the affected platforms reset their user's passwords of their users after an exposure, but this does not alert the users about the risk of using the same credential on other platforms. Hence, there is a demand for *Compromised Credential Checking* (C3) tools [35] that allow users to check whether their credentials are breached or not.

Popular password managers already integrate C3 services: 1password uses *HaveIBeenPwnd* (HIBP) [30, 45] and LastPass uses ENZOIC [20, 21]. These schemes offer up to four different query types: querying the client's username or password, the service's domain, and the combinations of the client's username and password. Thomas et al. [48] conclude that querying the username/password combination is the best option due to the user-friendliness, as password-only queries would alert users too often and the other two options are too vague (cf. [48] for further discussions). Recently, Thomas et al. [48] published their *Google Password Checkup* (GPC) tool as a Google Chrome extension that is the first C3 service secure against malicious clients (a variant of this is now integrated in Chrome, see below). They achieve this with the help of a *Private Set Intersection* (PSI) protocol that enables one party to privately check if her input is in the set of the other party (actually this is a variant of PSI where one input set consists of a single element only). To optimize efficiency, all these tools run the PSI protocol only on a small subset of the entire database, where the elements have the same prefix of the hashed credentials. For this, the hash prefix is leaked to the server. However, the hash prefix can be used for a credential stuffing attack on the user's anonymity as the server learns in which subset the credentials would be located [35]. A PSI protocol on the whole database would avoid such leakage, but it is too inefficient for large-scale databases.

This attack is not only theoretical. Li et al. [35] showed that knowledge of the credential's hash prefix suffices to compromise up to 86% of the leaked accounts within 1000 attempts (even up to 73% of the accounts that are not included in a data breach). To protect the user's sensitive information, they provide two new C3 protocols from which one still has the leakage problem that enables credential stuffing attacks. The other protocol was proposed in parallel by Thomas et al. [48] and does leak no information about the user's password since the subset is identified by a prefix of the hashed username. This protocol, however, has the disadvantage that the user's anonymity is even more vulnerable since the adversary learns information about the username. Moreover, the protocol can only be deployed for applications where the user can check the existence of its username/password combination in a data breach, while more security-aware users aim to check, if their passwords are attacked (even if the username is not included). In Aug. 2020, Google integrated and enabled by default this protocol in their Chrome web browser [11]. They hold a database of four billion leaked credentials and their deployed protocol leaks a three byte hash prefix of the username.

Thomas et al. [48] and Li et al. [35] both suggest to use *Private Information Retrieval* (PIR) to hide the hash prefix, which yields perfect anonymity, i.e., the C3 protocol does not allow to identify the user. However, [48] and [35] observe that current PIR techniques are not efficient enough to be operated in a real-world deployment. In this work, we show how to build and use highly efficient multi-server PIR for use in C3.

1.3 Setting and Applicability

Our model includes multiple servers, and guarantees security as long as there is no collusion of more than some threshold number of the servers. The usage of multiple servers seems crucial for ensuring both scalability and security. In fact, all large-scale C3 systems with a single server send to the server partial information about users credentials, which, as was shown by Li et al. [35], might compromise a large fraction of the users.

The assumption that servers do not collude with each other might not be credible by the public if all servers are run by the same entity (such as Google). Therefore, servers must be operated by multiple entities which are trusted not to collude. While this is a standard assumption/requirement in the cryptographic literature (e.g., for MPC protocols, multi-server PIR and threshold crypto), it is unclear if this assumption always makes sense from a business perspective. There are however, recent examples where companies are deploying services whose security depends on non-colluding servers. For example, Apple and Google run their exposure notification system for COVID-19 contact tracing via the Prio system [12] that is operated by the Internet Security Research Group (ISRG), which also runs the Let's Encrypt certificate authority and is therefore an entity which is trusted by Internet users, as well as the National Institutes of Health (NIH).[1] The additional servers can be run by organizations with a privacy-centric mission, or by different companies which would like to collaborate in order to provide a service to the public.

The recent Apple/Google collaboration on an API for COVID-19 contact tracing is an example of a collaboration between companies (in a different domain) which was unimaginable until recently. This collaboration shows that two competing companies can have a strong mutual interest in offering privacy-preserving services to their users for real-world applications. Apple introduced in iOS14 It is very unlikely that any of these corporations would attempt to use a leakage in a C3 system to steal a user's password. However, as more users are becoming concerned about their privacy, companies might want to provide users with the strongest possible privacy that can be offered using multi-server PIR. Another motivation for participating companies, might be their desire not to be liable for knowledge of unnecessary private user data, or the fear that company insiders could try to learn such information.

1.4 Our Contributions

We propose, implement, and benchmark CIP-PIR, an efficient multi-server PIR protocol that is designed for large-scale applications operating multiple GB large databases and outperforms recent efficient PIR implementations like PIR based on function secret sharing [7]. Furthermore, CIP-PIR can be used by companies who want to provide privacy-preserving services to their customers as the underlying cryptography is so simple that even non-experts can be convinced of its security and correctness. For this, we design a strong new PIR model called client-independent preprocessing, which allows for the first time very efficient offline preprocessings completely *independent* of the client, i.e., the PIR servers do not even need to know the clients for the preprocessing. Our main contributions are summarized as follows:

Client-Independent Preprocessing PIR Model (§2.1). In the multi-party computation (MPC) literature the preprocessing model has prevailed as it gives tremendous speedups for the online computation by precomputing expensive cryptographic operations of the same type, ideally in parallel (e.g., somewhat homomorphic encryption in SPDZ [32]). Today, this model is common for many efficient state-of-the-art MPC protocols. In concurrent and independent work, the preprocessing model was applied to multi-server PIR in [33]. In that work a server interacts with the client in an offline phase to precompute some hints that are later used in the online phase. We go one important step further: our *Client-Independent Preprocessing* (CIP) PIR is *client-independent* and hence can be performed even before knowing the client(s). This allows local and parallel preprocessing across *all* clients with significant speedups.

CIP-PIR Protocol (§3.2). The first PIR scheme in our new CIP PIR model is called CIP-PIR that is based on the very simple RAID-PIR scheme by Demmler et al. [15, 16] and moves a part of the client's query generation to the server sides. This costs an additional round trip, but allows for very efficient offline preprocessing without involving the client. Moreover, the preprocessing phase can easily be batch-processed without waiting for numerous client requests resulting in faster amortized preprocessing and hence total time. We note that our scheme (as the original PIR by Chor et al. [10]) has linear communication complexity, but this is only one bit per block.

Database Compression in PIR (§3.3). We design and implement two database compression techniques that are applicable to all PIR constructions. Our first compression technique is applicable to all database types and improves the storage by factor $1.2\times$. Our second compression technique is designed for special-purpose hash databases and reduces the storage and online runtime by factor $5\times$ for a false-positive probability of 2^{-20}. There are many real-world applications for PIR on hash databases including all private set inclusion PIR applications (e.g., medical and patient databases), C3, and epidemiological modeling. Moreover, the GPC protocol [48] can profit from both of our compression techniques to reduce the database size by factor $5.9\times$.

CPU and GPU Implementation (§4). We implement our CIP-PIR protocol in a highly efficient manner in C++. Our parallel implementations for CPUs and GPUs are of independent interest as they are applicable also within other multi-server

[1]See, for example https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf.

PIR schemes, e.g., the linear XOR operations of PIR based on function secret sharing [7,8] after expanding the PIR query.

Our CPU implementation uses Intel AVX-512 intrinsics to support XOR operations over 512 bits within one CPU cycle. Moreover, we massively parallelize the server computation using the OpenMP framework. We gain (using the same codebase) for $n=2$ servers up to $2\times$ better online runtime ($5\times$ better for $n=5$ servers and collusion threshold $t=5$) than RAID-PIR [16] and $1.2\times$ faster amortized preprocessing time for batch size $|Q|=1\,000$.

Our GPU implementation provides for the first time a multi-server PIR implementation on a GPU using Nvidia's CUDA platform [9,43]. We implement and benchmark two approaches for efficient parallelization: With the first one, all CUDA blocks together compute the answer of one query, while the second one batches incoming queries and all the CUDA blocks process one query, separately. We gain for $n=2$ servers up to $2.1\times$ faster online runtimes than our CPU-based CIP-PIR implementation, and $85\times$ faster amortized online and preprocessing runtime for batch size $|Q|=1\,000$.

2 Preliminaries and Background

2.1 PIR Background

In *Private Information Retrieval* (PIR), as introduced by Chor et al. in [10], a client wishes to learn one or multiple blocks from a public database $|DB|$, while using only sublinear communication, and hiding the details of the query from the database owner(s). We focus on *multi-server PIR* schemes since they have a substantially lower computational overhead. In multi-server PIR the database DB is split over $n \geq 2$ servers that are assumed not to collude. The client sends a request to each server and combines the answers to compute the result.

Definition 1 (classical PIR protocol) *A classical PIR protocol is a tuple of algorithms* (`Create`, `Request`, `Response`, `Combine`) *as described below:*

`Create` *is locally run once by the owner of the database. It takes as input some data array D, and outputs for each server $i \in [n]$ a database DB_i which is sent to the server. (Note that unlike the original PIR constructions of [10], each server might receive a different database/state).*

`Request` *is run by the client and takes as input the index idx of the data item to access and outputs a list of queries $(q_0,\ldots,q_{n-1})$, where q_i is sent to server i.*

`Response` *is run by each server i. It takes the query q_i as input and outputs an answer a_i based on the local database DB_i.*

The client collects the answers $a_0,\ldots,a_{n-1}$ from the servers and calls the `Combine` *algorithm that outputs the desired data $d = D[idx]$.*

The PIR scheme must satisfy the following definitions for correctness (stating that the client must be able to correctly compute the answer of its query), and for security (stating that a subset of the serves learns nothing about the query).

Definition 2 (Correctness) *For any database D, let $\texttt{Create}(D) = (DB_0,\ldots,DB_{n-1})$. For any $idx \in [0,|D|-1]$, let $\texttt{Request}(idx) = (q_0,\ldots,q_{n-1})$. For any server j, let $a_j = \texttt{Response}(DB_j,q_j)$. Then $\texttt{Combine}(idx,a_0,\ldots,a_{n-1}) = D[idx]$.*

Definition 3 (Security of multi-query PIR) *A PIR scheme with redundancy parameter/threshold* $1 < t \leq n$ *is called* secure *if any collusion of less than t servers does not learn any information about the query indexes.*

In more detail, let κ be a computational security parameter and let T denote any subset of less than t servers, and let the view *of server j denote DB_j and all queries q_j that this server receives. We require that for any data D, and any two same-length sequences of (possibly not all unique) requests $R = (idx_1,\ldots,idx_m)$ and $R' = (idx'_1,\ldots,idx'_m)$, no algorithm whose run time is polynomial in m and in κ can distinguish the view of the servers in T, between the case of the client using the requests in R, and the case of it using R'.*[2]

The original PIR protocol of Chor et al. [10] guaranteed unconditional security. Our protocol will ensure only computational security as is defined here, based on the basic security assumption that pseudo-random generators exist.

A PIR scheme must also satisfy *communication efficiency*, by guaranteeing that the overall communication is smaller than sending D itself, and is only $o(|D|)$.

2.2 Multi-Server PIR schemes

Multi-server PIR assumes that a subset of the n operating servers are non-colluding. Let us first give an informal description of the scheme by Chor et al. [10], on which most multi-server PIR schemes are based. First, the input data D is split into B blocks of size b each, which results in the database DB. If the client is interested in learning block i it sends to the first server a random B-bit string q_0, and sends to the second server a string q_1 which is equal to q_0 in all bits except for the i-th bit, in which the two strings are different. Each server computes the XOR of the blocks which correspond to '1' bits in the string that it received, and sends the resulting b-bit block to the client. The client then computes the XOR of the two strings it received. This result is equal to the i-th block. The total communication with each of the n servers is $B+b$ bits.

[2] We note that the original PIR definition only protects the privacy of the client and not the privacy of the server. Namely, it does not prevent the client from learning more than a single data item. This property is called "symmetric PIR" [24]. It can be ensured by encrypting each entry of the server using a key known only to the server, and letting the client run a single instance of an efficient oblivious pseudo-random function evaluation protocol (OPRF) in order to learn the decryption of only a single item [22,40].

The core idea of multi-server PIR directly follows from this protocol. The client sends messages to each of the n servers. Server i extracts from its message a B-bit query strings q_i. The XOR of the n query strings of all servers is a string of B bits, which are all zero except for a single '1' bit at the position corresponding to the block that the client aims to retrieve. In the online computation, each server i computes the XOR of all database blocks which correspond to '1' bits in q_i. Different PIR schemes differ in encoding the request on the client side and extracting the B-bit strings on the server side. We now describe how the client queries are encoded and extracted in RAID-PIR [15, 16] and in all PIR schemes based on function secret sharing [7, 8].

RAID-PIR [15, 16]. In RAID-PIR [15, 16] the B blocks are split into n chunks (recall that n is the number of servers). Each server receives $t \leq n$ chunks, and therefore stores t/n of the database (where t is the collusion threshold). Consequently, the client's queries are shorter and each server only XORs a smaller subset of the blocks. As before, the XOR of all n queries is equal to a B-bit string which is also zero except for a '1' bit at the block that the client wishes to learn.

Each block is included in a chunk of $k = B/n$ blocks, which is stored by t servers. A crucial observation that is used to improve performance is that for any specific block, out of the tk bits that instruct t servers what to do with this block, $(t-1)k$ bits can be pseudo-random and only k bits need to be explicitly set to ensure that the result of the XOR is correct. Therefore instead of sending a full length string to each server, the client can send to each server a seed that is used to compute a $1/t$ fraction of the string that the server must use. This cuts the communication from the client to the server by factor $t\times$. Performance can further be improved with a time-memory tradeoff that precomputes queries using the method of Arlazarov et al. [3] (known as the "method of the four Russians"), and optimizing the database layout to allow for parallel queries.

PIR based on Function Secret Sharing. PIR based on function secret sharing (FSS) as proposed by Boyle et al. [8] uses a distributed point function in order to encode the query of the client. A distributed point function $f_{\alpha,\beta}$ is a pair of functions f_0, f_1, where $f_{\alpha,\beta}(x)$ evaluates to β for $x = \alpha$ and to 0 for $x \neq \alpha$, and $f_{\alpha,\beta}(x) = f_0(x) + f_1(x)$ holds for all $x \in \{0,1\}^B$ and $B \in \mathbb{N}$ (the number of blocks in PIR). A function secret sharing on distributed point functions is instantiated by a client choosing two keys k_0 and k_1, defining two functions $\hat{f}_0(x) = f_0(k_0, x)$ and $\hat{f}_1(k_1, x) = f_1(x)$, and distributing these to two servers. When a client aims to retrieve block i in PIR, he generates a distributed point function $f_{i,1}$ and secret shares this to the two servers via function secret sharing. Server j then can extract the query by calculating $\hat{f}_j(x)$ for all $x \in \{0,1\}^B$, which then results in a B-bit string q_j. The XOR of these two strings q_0 and q_1 results in a zero B-bit string with only one '1' at position i. In addition, FSS-PIR is mostly applicable to the two-server setting.

FSS-based PIR is the first multi-server PIR scheme that has logarithmic upload communication complexity, while the download depends on the chosen blocksize b. Although the communication complexity of our new CIP-PIR scheme is sublinear, the total online runtime of CIP-PIR improves over FSS-PIR for large databases (cf. §5.2.2)

2.3 CUDA

Nvidia's Compute Unified Device Architecture (CUDA) [9, 43] is a framework that allows to program GPUs to compute Single Instruction, Multiple Data (SIMD) instructions and to support direct memory access. It is a popular architecture for highly parallelized programming run by a large number of threads each performing simple instructions from CUDA's own instruction set architecture called *PTX ISA*. All threads have access to a global memory that has a high storage capacity, but takes at least 400 clock cycles per memory access.

The threads are grouped into multiple CUDA blocks each having a shared memory among its threads. As accesses to the shared memory are very efficient, we can apply a technique called *coalescing* to bundle the global memory accesses of multiple threads into a single access within a block.

2.4 Threat Model

Throughout this work, we evaluate PIR protocols relative to the following main threat model: a coalition of malicious PIR servers try to learn information about the client's query. In this threat model, servers can deviate from the protocol description in order to learn details about the PIR query. This is obviously critical, since the contents of the query depends on the underlying application, which we aim to protect by using PIR. All PIR protocols are designed to hide queries from coalitions of less than t servers, where t is the collusion threshold which should obviously be as close as possible to the total number of servers. (Clearly, if all PIR servers in multi-server PIR schemes collude, they are able to reconstruct the client's plain query.)

3 Private Information Retrieval Extensions

We introduce our new Client-Independent Preprocessing PIR model (§3.1), describe our CIP-PIR scheme (§3.2), and present our new PIR database compression technique (§3.3). We give a summary of the RAID-PIR scheme from [15, 16] in §A, the security proof of CIP-PIR in §B, its complexity analysis in §C, and efficient database updates in our full version [27, § D] .

3.1 PIR with Client-Independent Preprocessing (CIP-PIR)

Previous works on PIR, such as [6, 16], improve the online computation for the server with a time-memory tradeoff that merges and precomputes once in a setup phase parts of the database. Then, during the Response method, the servers only have to combine the precomputed parts depending on the query q_i. Our idea is fully compatible with this time-memory tradeoff, but goes one significant step further.

We split the preprocessing into two parts - *database preprocessing* and *client-independent preprocessing*. The database preprocessing is a one-time precomputation step in the setup phase that maps the database into a state that enables the servers to compute their answer more quickly as described in [6, 16]. We use this known optimization in our implementation but do not include it in our presentation for simplicity. In addition, we introduce the client-independent preprocessing which is a client-independent routine in the preprocessing phase that precomputes concrete parts of the server's answer for one query which can be used only once. The client-independent preprocessing is of course independent of the contents of the query and can be computed before it is received by the server. In the following, we define our new *client-independent preprocessing* (CIP) PIR model, which goes beyond the Offline/Online model of [13] as it computes the preprocessing/online phase *without* involving the client.

Definition 4 (PIR in the CIP model) *A PIR scheme in the CIP model is a tuple of algorithms* (Create, Preprocess, Request, Response, Combine)*. The protocol is shown in high-level in Fig. 1.*

The Create *and* Combine *algorithms are exactly the same as in the original PIR model from §2.1.*

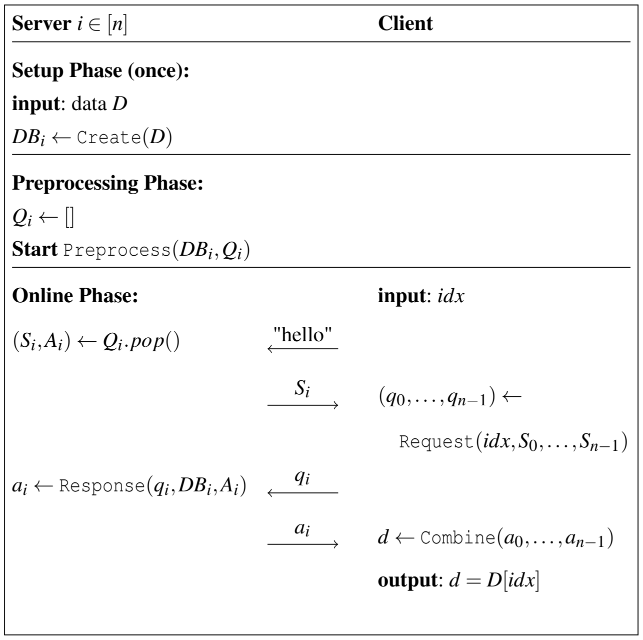
Each server i locally runs the Preprocess *algorithm in a parallel thread that can be started and paused. This algorithm takes as input the database DB_i and adds query-specific tuples (S_i, A_i) to the queue Q_i until it is full or the thread is interrupted. The run is paused until there is new space for more values in Q_i. S_i is a short seed and A_i is a part of the server's answer for the i-th query that depends only on S_i, but not on the query q_i, i.e., A_i is* independent *of idx.*

After the client sends its "hello" message to the servers, each server i pops one pair (S_i, A_i) from Q_i and sends S_i to the client. The Request *algorithm takes as input the index idx of the data item to access, and seeds $S_1, \dots, S_n$ obtained from the n servers, and generates queries $q_1, \dots, q_n$.*

Each server i calls the Response *algorithm on input DB_i, A_i and the received query q_i to return its answer a_i.*

3.2 Our CIP-PIR Protocol

We now describe the details of our new CIP-PIR protocol, which is the first PIR protocol in the new client-independent



Server $i \in [n]$
Client
Setup Phase (once):
input: data D
$DB_i \leftarrow$ Create(D)
Preprocessing Phase:
$Q_i \leftarrow []$
Start Preprocess(DB_i, Q_i)
Online Phase:
input: idx
$(S_i, A_i) \leftarrow Q_i.pop()$
"hello"
S_i
$(q_0, \dots, q_{n-1}) \leftarrow$ Request$(idx, S_0, \dots, S_{n-1})$
$a_i \leftarrow$ Response(q_i, DB_i, A_i)
q_i
a_i
$d \leftarrow$ Combine$(a_0, \dots, a_{n-1})$
output: $d = D[idx]$

Figure 1: Messages in client-independent preprocessing PIR (CIP-PIR). Client communicates with all n servers $i \in [n]$ in parallel.

preprocessing PIR model. We give the security proof of CIP-PIR in §B and its complexity analysis in §C.

RAID-PIR [15, 16] improves over Chor et al.'s scheme [10] in terms of communication by using seeds, and in terms of online computation by requesting each server to only touch a subset of the database. The second improvement reduces security as the number of servers that are allowed to collude is reduced from $n-1$ to $t-1$, where $2 \leq t \leq n$ is the threshold.

The general idea of our CIP-PIR scheme is to use the RAID-PIR scheme, but instead of having the client choose the seeds in the Request algorithm, let the *servers* choose the seeds in the Preprocess algorithm. This enables the servers to compute in advance $(t-1)/n$ of the XOR operations, and complete in the online phase only the remaining $1/n$ XOR operations. The online computation is therefore independent of the collusion threshold t. Then, the servers give the seed to the client and the protocol proceeds as before.

Create (Algorithm 1). In the setup phase, the input data D is split into B blocks of b bits each. These blocks are grouped into n chunks of $k = B/n$ blocks. $chunk_i$ denotes the flip chunk of server i and is the first chunk in the database DB_i of server i. Note that all servers hold the same database, but the order of their chunks differs.

Preprocess (Algorithm 2). As depicted in Fig. 1, the server permanently computes (seed, value)-pairs (S, A) and pushes them to its local queue Q_i. The seed S is expanded to

Algorithm 1 `Create` of CIP-PIR

```
input: $D$                                              ▷ data $D$
$(block_0,\ldots,block_{B-1}) \leftarrow D$              ▷ split $D$ into $B$ blocks
$k \leftarrow B/n$                                       ▷ # blocks per chunk
for all $i \in [n]$ do
    $chunk_i \leftarrow block_{ki}|\ldots|block_{ki+k-1}$
end for
for all $i \in [n]$ do
    $DB_i \leftarrow chunk_i|chunk_{i+1 \bmod n}|\ldots|chunk_{i+t-1 \bmod n}$
end for
return $DB_0,\ldots,DB_{n-1}$                            ▷ database for each server
```

a $k(t-1)$ bit query q via a `PRG`. The precomputed value A is then the XOR of all blocks from the non-flip chunks whose corresponding bits in the query q are set to 1. Since each server has $t-1$ non-flip chunks among the n chunks, the preprocess algorithm precomputes $(t-1)/n$ of the database s.t. only $1/n$ of the database is left for the `Response` algorithm (Algorithm 4) in the online phase.

Algorithm 2 `Preprocess` of CIP-PIR

```
input: $DB_i, Q_i$                                       ▷ database $DB_i$, queue $Q_i$
$(chunk_0,\ldots,chunk_{t-1}) \leftarrow DB_i$ , $k \leftarrow B/n$
$DB \leftarrow chunk_1|\ldots|chunk_{t-1}$               ▷ all but first chunk
while $Q_i$ is not full do
    $S \leftarrow_{\$} \{0,1\}^{\kappa}$
    $q \leftarrow PRG(S_i, k(t-1))$                      ▷ non-flip chunk
    $A \leftarrow q \cdot DB$              ▷ XOR blocks of $DB$ corresponding to $q$
    $Q_i$.push$(S,A)$                                    ▷ push seed/value pair $(S,A)$
end while
```

`Request` (Algorithm 3). The `Request` algorithm generates the flip chunk q_i for each server i depending on the server's seeds S_i, and the requested block index idx. Firstly, the main query q is built by setting the idx-th bit of a B-bit

Algorithm 3 `Request` of CIP-PIR

```
input: $idx, S_0,\ldots,S_{n-1}$                 ▷ index $idx$, seeds $S_0,\ldots,S_{n-1}$
$q \leftarrow 0^B$                     ▷ the main query consists of 1 bit per block
$q[idx] = 1$, $k \leftarrow B/n$                         ▷ # blocks per chunk
for all $i \in [n]$ do
    $v \leftarrow PRG(S_i, k(t-1))$                      ▷ pseudo-random bits
    $u \leftarrow \min(i+t, n)$, $w \leftarrow \max(i-t+1, 0)$
    $q[ki : ku-1] \leftarrow q[ki : ku-1] \oplus v[0 : k(n-u+1)-1]$
    $q[0 : kw-1] \leftarrow q[0 : kw-1] \oplus v[k(n-u+1) : kt-1]$
end for
for all $i \in [n]$ do
    $q_i \leftarrow q[ik : (i+1)k]$
end for
return $q_0,\ldots,q_{n-1}$                              ▷ query for each server
```

vector to 1. Then, the client expands the seeds S_i to a $k(t-1)$ bit sub-query v which covers all non-flip chunks of server i. Server i has $u-i$ chunks to the right ($ku-1$ points to the last block of the u-th chunk, which is the last chunk of the database if $i+t > n$) and the first w chunks of the database (w can be 0 if $i+t \leq n$). The expanded sub-query v is XORed to the main query at the respective chunks. Finally, the resulting query q is split into n sub-queries q_i that are the flip chunks for each server i.

Algorithm 4 `Response` of CIP-PIR

```
input: $q_i, DB_i, A_i$   ▷ query $q_i$, database $DB_i$, precomputed block $A_i$
$(chunk_0,\ldots,chunk_{t-1}) \leftarrow DB_i$           ▷ $chunk_0$ is flip chunk
$a_i \leftarrow A_i \oplus q_i \cdot chunk_0$
return $a_i$                                             ▷ answer of server $i$
```

`Response` (Algorithm 4). Server i extracts its flip chunk from the database ($chunk_0$ in DB_i) and XORs all blocks corresponding to bits set to 1 in the query q_i. The result is XORed to the precomputed block A_i from the `Preprocess` algorithm and returned to the client. Here, only one chunk, i.e., $1/n$ of the database is touched, whereas the remaining $(t-1)/n$ of the database are already precomputed in block A_i.

Algorithm 5 `Combine` of CIP-PIR

```
input: $a_0,\ldots,a_{n-1}$                              ▷ answers of the servers
$d \leftarrow \bigoplus_{i=0}^{n-1} a_i$
return $d$                                               ▷ data block $d$
```

`Combine` (Algorithm 5). The XOR of all server answers a_i is obtained as $d = D[idx] = \bigoplus_{i=0}^{n-1} a_i$.

For each query, the server pops a pair and sends seed S_i to the client after obtaining its initial *"hello"* message.[3]

Communication Comparison. Table 1 compares the *online* communication complexities of our CIP-PIR scheme with the recent FSS-PIR [7, 8] and Online-Offline (OO)-PIR [33] implementations for downloading a $\hat{b} = \sqrt{|DB|/8n}$ block as required for a PIR-based C3 protocol. The concrete values shown in this table are computed for a $|DB| = 16$ TB database, which is three orders of magnitude larger than GPC's C3 database with all of our compression techniques. For a fair comparison, we assume that these compression techniques were applied to the other PIR schemes as well.

The novel sharing of the client's request via a distributed point function makes the upload very cheap in FSS-PIR, while

[3] An outer protocol must ensure that clients cannot run a denial of service attack by just sending many *"hello"* messages which would quickly drain the server's queue. This can be done with proper rate limiting, e.g., using client puzzles [31].

Scheme	Communication		Concrete (Upload + Download)
	Upload	Download	($\|DB = 16$ TB, $n = 2$, $\kappa = 128$, $b = 1$ KB)
CIP-PIR [this work]	$n\sqrt{\|DB\|/8n}$	$n\kappa/8 + n\sqrt{\|DB\|/8n}$	4 096,0 KB
FSS-PIR [7]	$\kappa(\log_2(\|DB\|/128)+2)$	nb	2,6 KB
OO-PIR [33]	$2(\kappa \log_2 \|DB\|+1)\log_2(\|DB\|)$	$4b$	64,6 KB

Table 1: Online communication comparison of our CIP-PIR scheme with FSS-PIR [7] and Online-Offline (OO)-PIR [33] on a $|DB|$=16 TB database with $n = 2$ servers and security parameter $\kappa = 128$ bit. We set the blocksize $b = 1$ KB for FSS-PIR and OO-PIR.

the download complexity depends on the chosen blocksize b. However, for some applications like C3 [48] and epidemiological modeling [28], where large data items need to be downloaded and thus the blocksize b increases, the communication complexity of FSS-PIR approaches to CIP-PIR, which has about the same amount of upload and download for the optimal blocksize. We see the same observation for OO-PIR, which is very efficient for retrieving single bits or small data entries, but its download can even increase that of CIP-PIR for large blocksizes b.

3.3 Database Compression in PIR

In the following, we show how to compress the database in PIR schemes by (a) adapting a database compression technique which was used in [46] in the context of a private membership test, and (b) by shorting the hash values in the database. We compute the optimal blocksize where the amount of uploaded and downloaded data is nearly equal and thus the total communication is minimal. We call this optimization *PIR with Database Compression*. It can be applied to any PIR scheme that is based on blocks.

Storing the Differences. The idea of the technique of [46] is to first sort the entire database before it is divided into blocks. Assume that a block has the entries $(e_1,\dots,e_m)$. 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, namely only store $(e_1, e_2 - e_1, e_3 - e_2, \dots, e_m - e_{m-1})$. It is easy to see that the length of the differences is smaller than the length of the entries.[4] This compression technique can be applied to any PIR scheme that is based on blocks.

Since the client only retrieves a single block of the database, and decompressing the entire database on the server side would be very inefficient, we apply this compression technique independently to each block of the database. Therefore the client does not need to know any data except for the retrieved block to decompress the block. For a better compression, we increase the blocksize b. Thus, we use fewer blocks while the blocks become larger but are stored and sent in a compressed way. Using larger blocks induces a tradeoff: it increases the communication from the servers to the client, but reduces the communication from the client to the servers.

[4]Suppose that a set contains m items from a domain of size N. Storing the items themselves requires $m \log N$ bits. On the other hand, if the items are evenly distributed, as is the case when they are generated as outputs of a hash function, then the average distance between two successive items is N/m, and we need to store only $O(m(\log N - \log m))$ bits.

Shorter Hashes. In C3 [48], each compromised credential is represented as a 32 bytes hash prefix of H, which results in a database of $|DB| = 32 \cdot N$ bytes, where N is the number of compromised credentials. Since we only need to check for equality, we can apply a trick from the PSI literature [17, 41, 42] and only use the first $40 + \log_2 |DB|$ bits of $\mathcal{H}(H)$ instead. (The probability of a collision between the hash of the user credential and any other hash is therefore only 2^{-40} and hence negligible.) For a database of size 5 billion entries this cuts the size of each entry down to only 73 bits, which reduces the database size by factor $3.6\times$. Using a more relaxed bound on the false error probability would result in even shorter values, e.g., a bound of 2^{-20} (meaning that one in a Million users gets a false warning) requires only 53 bits and results in a $5\times$ reduction of the database size. This compression techniques can be used for any PIR database whose entries consists of blocks and hash values are used to check for equality.

Optimal Blocksize. Let b be the size of blocks after compression. The total communication per server for CIP-PIR is

$$\mathcal{C}(b) = \frac{B}{n} + \kappa + b = \frac{|DB|}{bn} + \kappa + b, \qquad (1)$$

where the first term is the size of the information sent from the client to the server, and the last two elements are the size of the data sent from the server. One can easily show by derivation that $\mathcal{C}(b)$ has its local minimum at $\hat{b} = \sqrt{|DB|/n}$. Later in §5.2.2, we will show that this reduces the size of the DB by factor $1.2\times$ compared to the uncompressed database without any further optimization. We also show that the theoretically computed values almost perfectly match with the measured communication. Note that Eq. 1 only calculates the communication for one server. We can multiply $\mathcal{C}(b)$ by n to get the total communication among all servers. The upload and download are both sub-linear in $|DB|$.

4 GPU-Accelerated Multi-Party PIR

In the PIR literature so far, GPUs were used to accelerate heavy computations (relying on cryptographic hardness assumptions) in single-server PIR [14, 37]. In multi-server PIR, the computations are mainly cheap independent XOR operations for which modern processors require only one clock cycle for a 512 bit block. Hence, for multi-server PIR, it was unclear if the overhead induced by moving data between CPU and GPU as well as managing CUDA blocks pays off or if it is more efficient to directly compute the XORs on the CPU. In this work, we show for the first time that GPUs can accelerate such cheap operations in multi-server PIR. For this, we make use of the efficient thread management capabilities of Nvidia's CUDA architecture (cf. §2.3). We present two approaches for accelerating the huge amount of XOR operations of multi-server PIR in §4.1.

In CIP-PIR (cf. §3.2), especially the offline phase can massively profit from outsourcing it to GPU clusters by batching the computation of multiple (seed, value)-pairs. This massively reduces the internal memory shifting operations on the GPU which are necessary for loading into the shared memory the database chunks that are currently computed. In §4.2, we demonstrate how GPUs can substantially improve the amortized runtime by batching multiple queries.

4.1 GPU-Acceleration of XOR Operations

The massive number of XOR operations are the main cost factor of multi-server PIR. In this section, we demonstrate two approaches for parallelizing these computations efficiently on a GPU using Nvidia's CUDA architecture (cf. §2.3).

All Compute One (ACO). In this approach, all CUDA-blocks simultaneously compute a single (seed, value)-pair together by looping over the query and one word[5] of the output is computer per thread. As the output consists of b bytes (blocksize), we need $C = \lceil b/T \rceil$ CUDA-blocks, where T denotes the number of threads in a CUDA-block. If the GPU has more than $T_{pair} = C \cdot T$ threads, we can compute multiple pairs in parallel, i.e., the maximum number of pairs that can be computed in parallel is T_{max}/T_{pair}, where T_{max} denotes the number of threads on the GPU.

In-Register. In the in-register approach, each CUDA-block with T_{max} threads computes one pair, where each thread is responsible for $\lceil b/T_{max} \rceil$ bytes of the b byte output. As the thread's registers have the fastest memory access speed, we can accelerate the computation by storing the intermediate results in these registers. The maximum number of pairs that can be computed in parallel is not clearly defined since GPUs with CUDA Compute Capability 3.0 or higher can handle up to $C_{max} = 2^{31} - 1$ CUDA-blocks. However, the maximum number of threads T_{max} und the GPU are the limiting factor of this approach as well. Thus, one can not naïvely set the amount of CUDA-blocks to the maximum value C_{max} since only a few threads would compute on a single pair simultaneously. Instead, we dynamically set the number of threads per CUDA-block T and the number of CUDA-blocks C depending on the blocksize b to significantly improve the performance.

[5]The word size depends on the GPU's architecture (4 bytes for our GPUs).

4.2 Amortized Query Preprocessing

Batching multiple queries was already used in computational PIR schemes [2, 23], but required waiting for multiple client queries in IT-PIR schemes [1, 13] to collect several client requests which increases online latency. We can now batch multiple queries in the preprocessing phase without increasing online latency in CIP-PIR as the computations are completely independent of the client.

A main performance bottleneck of GPU-accelerated PIR computation is that multiple portions of the database must be copied into the GPU's memory, which costs many clock cycles (cf. §2.3). If we instead compute M (seed, value)-pairs in parallel, we can amortize these times for copying the database portions to the GPU among all M pairs. Consequently, the total runtime of CIP-PIR consisting of the online and the amortized preprocessing phase for a single query, is faster than the online phase of a RAID-PIR query, as batching multiple queries in RAID-PIR requires to wait for multiple incoming queries (which obviously also takes extra time). Hence, the amortized runtime in CIP-PIR improves over RAID-PIR by factor $1.3\times$ for CPU and by up to factor $53\times$ for GPU (cf. §5.2.1).

5 Implementation and Benchmarks

We implemented a CPU-based (cf. §3.2) and a GPU-accelerated version (cf. §4) of our CIP-PIR protocol in C++, as well as a CPU based version of FSS-PIR [8] using the recent improvements on distributed point functions by Boneh et al. [7]. We give the implementation details in §5.1 and runtimes in §5.2.

Use-Case. As use-case for our experiments, we use Compromised Credential Checking (cf. §1.2), where the client obliviously retrieves a block from the database and checks if her hash computed from her username and password is contained in it. The 32 byte hashes are compressed to 8 bytes (cf. §3.3).

5.1 Implementation

Our implementation consists of three components: the database generation, the server, and the client. We summarize the details of the first two components next.

Database Generation. Our database consists of pseudorandom values, which simulates a real-world deployment of C3 related applications (cf. §1.2), as hashed values are pseudorandom as well and hence have the same distribution. The database is stored in the RAM of the OS and the server's GPU memory. We created databases up to 3.5 billion entries which suffices to cover the password breaches in Collection 1-5 [26].

Server. The online server's main task is to answer the client's queries by computing the corresponding values based on the precomputed (seed, value)-pairs. In the CPU-based implementation, we use Intel AVX-512 intrinsics to enable XOR operations over 512 bits with a single CPU instruction. On top of this, we parallelize this approach using OpenMP. Since the server does not need the whole query to start the answer computation, we implemented a pipelining approach that directly processes the query while it still receives the client's query.

5.2 Benchmarks

We benchmark our CIP-PIR schemes as follows: In section §5.2.1, we benchmark the amortized preprocessing runtime for various blocksizes b. In section §5.2.2, we benchmark our CPU-based and GPU-accelerated CIP-PIR implementations and compare them with RAID-PIR [16] on the same codebase and the single-server SealPIR [2].

Experimental Setup. For the benchmarks, we use the following Amazon AWS instances: For the GPU-accelerated CIP-PIR and FSS-PIR servers, we use p3.2xlarge instances each having an NVIDIA Tesla V100 yielding a computational power of 7 TeraFLOPS and 16 GB of HBM2 memory with a bandwidth of 900 GB/s and a wordsize of 4 bytes. The machines have 8 vCPUs and 61 GB RAM which is sufficient for our use-case because we can load databases up to the size of the GPU memory, i.e., 16 GB in total. For the CPU-based PIR implementations, we use c5.24xlarge instances which deliver a high performance for compute-intensive workloads. These instances feature 2nd generation Intel Xeon 8000 series processors with a clockspeed of up to 3.6 GHz, 96 vCPUs and 192 GB RAM in total to provide a fair comparison to the GPU based approach. When writing this work, the p3.2xlarge costs 3.823 USD per hour and the c5.24xlarge 4.656 USD per hour, so that the GPU-accelerated instance is roughly 20% cheaper. For the client, we use a t2.large instance, which has 2 vCPUs installed and 8 GB of RAM. Between client and servers, we measured a network bandwidth of 1 GBit/s. We always give average execution times over 10 benchmark runs.

5.2.1 Preprocessing Phase

We first benchmark the amortized runtimes of the preprocessing phase of our CIP-PIR protocol.

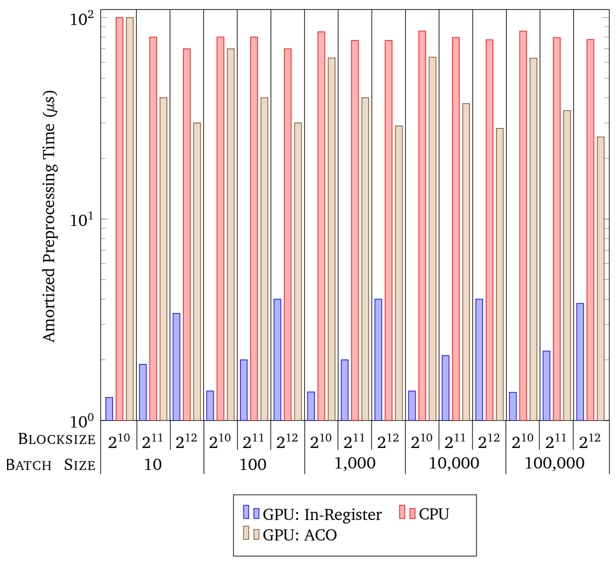


Figure 2: The amortized preprocessing time per (seed, value)-pair of our CPU-based and two GPU-accelerated implementations. We use $n = 3$ servers, a database of 1 million entries and an entry size of 8 bytes. The blocksize is given in bytes.

Influence of blocksize and number of pairs. In this benchmark, shown in Fig. 2, we measure the amortized preprocessing runtime for the CPU-based implementation and both parallelization techniques of the GPU-accelerated implementation from §4.1. For this benchmark, we used a 8 GB database consisting of 1M entries of 8 bytes and $n = 3$ servers, i.e., 2/3 of the whole database is processed in the preprocessing phase. The amortized runtimes grow linearly with the database size (not shown in our benchmarks). We give benchmarks for various blocksizes and number of simultaneous computed (seed, value)-pairs.

CPU: The amortized runtime has no high impact on the CPU-based implementation. The speedup of factor $\approx 1.3\times$ over the non-batched execution is only measurable until all threads are occupied, but afterwards, the performance falls back to a factor of $\approx 1.1\times$ improvement. However, the CPU-based implementation scales better for larger blocksizes, so it is a natural choice to set the blocksize b to the optimal blocksize $\hat{b}$ that yields the best communication overhead (cf. §3.3).

ACO: In the ACO parallelization technique (cf. §4.1), all CUDA-blocks compute one (seed, value)-pair together. We see a significant amortized runtime improvement compared to the CPU-based implementation. This improvement grows with the blocksize since the ACO technique scales very well with larger blocksizes: Each thread needs to XOR a higher number of blocks when choosing smaller blocksizes, since the ACO approach uses the whole GPU computational power to evaluate one seed after another. As long as the GPU's threads are not occupied, we observe a massive performance

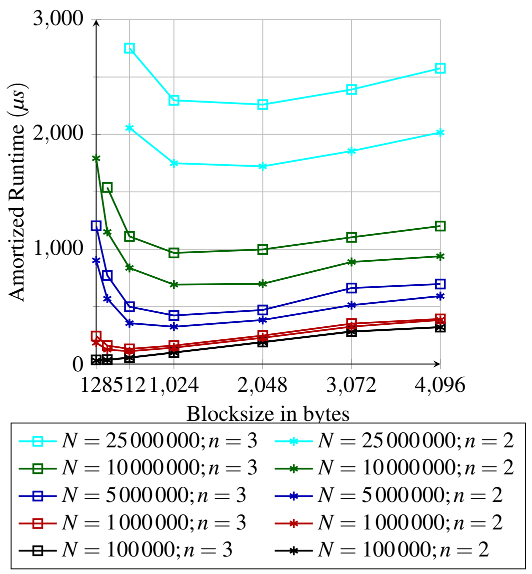


Figure 3: Runtimes for various blocksizes on the GPU-accelerated implementation using the in-register approach. Different colors indicate the number of 8 byte entries in the database, whereby the marks show the amount of servers ($n = 2$ and $n = 3$) used in the corresponding experiment.

improvement as each thread processes one byte of the block. Unfortunately, we do not gain further amortized runtime improvements for more than batch size $|Q| = 1\,000$ since the ACO technique scales only linearly with the number of pairs. However, the ACO approach improves the CPU-based implementation up to factor $2.6\times$ for a batch size of $|Q| = 1\,000$

In-Register: Our most optimized approach called in-register, where the threads compute on the values inside their registers (cf. §4.1), shows a clear improvement over the ACO-based by up to factor $63\times$ and the CPU-based implementations by up to factor $\approx 85\times$ for a batch size of $|Q| = 1\,000$. This approach outperforms the amortized total runtime including preprocessing and online time of RAID-PIR by factor $\approx 53\times$. With a higher batch size and larger blocksizes the speedup factor is still $18\times$. Aside from minimizing the memory accesses with high costs, the in-register approach benefits from choosing the parameter for the CUDA-blocks dynamically based on the blocksize and the hardware specifications. We see that the amortized runtimes grows linearly with the number of simultaneously computed pairs. However, it is the only approach where increasing the blocksize has a negative impact on the performance due to the overhead of XORing more bytes per block. An optimization that is left for future work is pipelining where already computed results are copied to the server's main memory while performing further precomputations s.t. the cost of data transmissions can be hidden almost completely.

Best blocksize for in-register approach. In this benchmark, shown in Fig. 3, we measure the preprocessing runtime of the in-register-based implementation for various blocksizes and database sizes, for $n = 2$ and $n = 3$ servers to investigate whether an optimal blocksize for this approach exists.

We see in Fig. 3 that each database - except for the smallest one with 100 000 entries - shows similar characteristics for the in-register implementation: the overhead with too small blocksizes is huge, but decreases exponentially to the optimal blocksize. Afterwards, the runtime increases nearly linearly with the blocksize. It is interesting to see that the optimal blocksize scales only marginally with the size of the database, e.g., with $N = 1$ million entries the optimal blocksize is 512 bytes whereas with $N = 5$ million entries it is 1 KB.

As the in-register approach is also used during the online-phase of our GPU based FSS-PIR implementation, we will constantly use 1 KB blocks for our further benchmarks which gives the highest experimentally deviated speedup for the online phase.

5.2.2 Setup and Online Phase

We compare our CPU-based CIP-PIR implementations (cf. §5.1) in C++, our reimplementation of FSS-PIR [7, 8] and RAID-PIR [15, 16] using the same codebase (including the parallelization and pipeline optimizations outlined in §5.1), the original Python implementation of RAID-PIR from [15, 16] in Python, and the publicly available single-server SealPIR implementation in C++ [2].

Setup Phase. In the one-time setup phase, a random PIR database is generated, sorted, compressed, and the precomputations related to the database are processed and written to a file. This phase is identical for RAID-PIR, CIP-PIR, and FSS-PIR, and its bottleneck is the precomputation using the method of Arlazarov et al. (the "four-Russians" algorithm) [15], which divides the database into equal sized groups and precomputes all possible query combinations for each individual group. In our implementation, we choose a group size of 8, which requires precomputing 255 combinations for each group. For our largest database of size 25 GB, this took roughly 84 minutes. The optimal blocksize, where the amount of upload and download data is almost the same, is $\hat{b} \approx 88$ KB, which results in roughly $B = 284\,000$ blocks. After compressing each block as described in §3.3, the blocksize is reduced by factor $\approx 1.2\times$ to $b \approx 73$ KB, which perfectly matches with the theoretical analysis.

Online Phase. The main difference between CIP-PIR and RAID-PIR is the amount of data each server has to touch in the online phase. Concretely, a CIP-PIR server touches $1/n$-th of the database, while a RAID-PIR server with threshold $2 \leq$

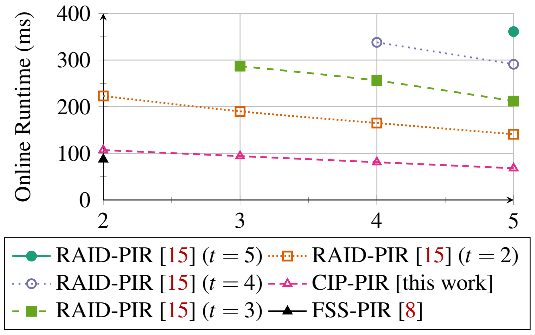


Figure 4: Online runtimes of our CPU-based PIR implementations for different number of servers n on a 500 MB database. Our implementations of the RAID-PIR [15, 16] and FSS-PIR [7, 8] protocols uses the same codebase as CIP-PIR. The threshold t of CIP-PIR is set to n.

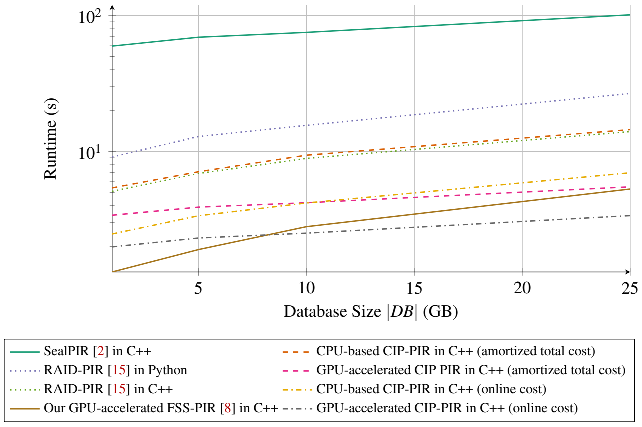


Figure 5: Runtimes of PIR implementations for different database sizes $|DB|$ and $n = 2$ servers ($n = 1$ for single-server SealPIR [2]). The threshold for all RAID-PIR implementations is $t = 2$. The batch size for the amortized total cost is set to $|Q| = 1000$.

$t \leq n$ processes t/n of the database.[6] Thus, the online time of our CIP-PIR protocol should improve over RAID-PIR by a factor of $t\times$. We observe this improvement in online runtime also in practice, as shown in Fig. 4, where 2-5 servers operate on a 500 MB database and achieve improvements of $\approx t\times$.

For a database of size 500 MB and $n = 2$ servers, our FSS-PIR implementation performs best. However, if we increase the number of servers to $n = 5$, our CIP-PIR protocol achieves a similar performance, as CIP-PIR only needs to process $1/n$ of the database in the online phase. So it is not surprising that the total online runtime decreases for CIP-PIR and RAID-PIR (where t/n of the database needs to be processed) decreases with the number of servers. Obviously, this behavior will not continue for larger number of servers n as the client's overhead of managing multiple connections becomes too high.

Communication. CIP-PIR and RAID-PIR have the same amount of communication (independent of RAID-PIR's collusion threshold t), but RAID-PIR has only a single round-trip while CIP-PIR has two round-trips. For a 500 MB database, the client uploads ≈ 17.6 KB and downloads ≈ 15 KB data with each server. An FSS-PIR client only needs to upload ≈ 500 byte (factor $35.2\times$ improvement) and downloads 1KB (factor $15\times$ improvement).

Comparison with other PIR implementations. In Fig. 5 we compare the online runtimes of several PIR implementations for varying database sizes and include the amortized total cost (online and preprocessing runtimes) for our CPU-based and GPU-accelerated CIP-PIR implementations. We compare the communication complexity of several PIR schemes in §C. For RAID-PIR [15, 16] we set the collusion threshold to $t = 2$ for best efficiency. CIP-PIR on a database size of $|DB| = 25$ GB improves over our RAID-PIR implementation by factor $\approx 2\times$ ($\approx 4.2\times$ for our GPU-accelerated CIP-PIR implementation with in-register, cf. §4.1). This matches what we would expect in theory as well. Most of the online runtime is spent on the server's huge number of XOR operations, which we halve in CIP-PIR. Our GPU-accelerated implementation improves over our CPU-based implementation by factor $\approx 2.1\times$. Moreover, our CIP-PIR implementation outperforms the original RAID-PIR implementation of [15, 16] in Python (without parallelization and pipelining optimizations) by factor $\approx 7.7\times$ ($\approx 16.2\times$ for our GPU-accelerated CIP-PIR implementation).

Our CPU-based CIP-PIR protocol is substantially faster than the state-of-the-art single-server PIR scheme *SealPIR* [2] by factor $\approx 16.8\times$ ($\approx 30.6\times$ for our GPU-accelerated implementation) as shown in Fig. 5. Single-server PIR schemes are based on expensive homomorphic encryption operations and the server needs to touch every bit of the database in order to gain no information about the queried block. Unfortunately, the current implementation of SealPIR does not implement networking, so we only measured the computation times, but already these were substantially slower than the total (computation + communication) times of (CIP-)RAID-PIR.

We chose a blocksize of $b = 1$ KB for our GPU-accelerated FSS-PIR implementation as this has a low communication overhead while our implementation performs best for this blocksize (cf. Fig. 3). For database sizes up to 8 GB, the GPU-accelerated FSS-PIR implementation outperforms our GPU-based CIP-PIR because, in addition to the higher communication costs, CIP-PIR requires another communication round. However, CIP-PIR can offset these additional communica-

[6]Note that the total amount of computation in CIP-PIR and RAID-PIR is exactly the same. CIP-PIR just shifts most of the computation costs to a preprocessing phase. The online communication is equal for both protocols and CIP-PIR just needs one more RTT, i.e., we have s slightly higher online communication time than RAID-PIR.

tion costs with savings in computational costs for database sizes $|DB| > 8$ GB, even outperforming the total runtime of FSS-PIR. This practical observation also matches with the theoretical analysis: Computational costs grow linearly with the database size, whereas communication costs increase only sub-linearly, i.e., the online computation becomes the bottleneck of all PIR protocols. On a high level, the online phase of FSS-PIR and CIP-PIR contain the same operations (excluding the query extraction from FSS-PIR), and differ in that CIP-PIR only does half as much work (e.g., XORing approx. 512 GB vs. 1024 GB for $|DB| = 1$ TB). When using the same codebase as we did in our benchmarks, CIP-PIR is faster then FSS-PIR as long as receiving the client's query (2 MB) is faster than XORing 512 GB. For $|DB| = 25$ GB, our GPU-accelerated CIP-PIR implementation improves over our GPU-accelerated FSS-PIR implementation by factor $\approx 1.6\times$. The amortized total runtime of our GPU-accelerated CIP-PIR implementation is worse for all of our benchmarked database sizes, however, we see for a database of $|DB| = 25$ GB that our FSS-PIR implementation only slightly (factor $1.04\times$) improves over CIP-PIR. For larger database sizes $|DB| > 25$ GB, we expect that even the amortized total runtime of CIP-PIR improves over FSS-PIR. This observation is caused by the fact, that less expensive memory movements need to be performed when processing $|Q| = 1000$ queries in parallel in the preprocessing phase.

6 Related Work

Multi-Server PIR. Chor et al. [10] introduced information theoretically secure PIR and gave first constructions that use n non-colluding servers where each server receives a query from the client and sends a response to it. Several subsequent works on multi-server PIR protocol [5,19,25,29] have the bottleneck of computing in the online phase many XOR operations over a large fraction of the database. The first multi-server PIR scheme with logarithmic communication complexity based on function secret sharing (FSS) via a distributed point function was shown by Boyle et al. [7, 8]. FSS-based PIR improves the upload communication by giving each server a distributed function share, where all shares together are expanded into the client's query. Afterwards, this scheme still computes XOR operations over the whole database which, as we show, is the main bottleneck also in Chor et al. [10]-based PIR for large databases, which we significantly improve in our work.

Most previous works on preprocessing PIR [6, 15, 16] perform an expensive one-time offline precomputation phase for database-dependent values reducing the online computation by a constant factor. As client-dependent offline phases are well-established practice in MPC, this model also found its way to the PIR literature. In a new work [13], the authors introduce a new PIR model called *Offline/Online* (OO)-PIR, where the servers and the client run a preprocessing phase before the client knows which database entry it wants to access. The main difference between their model and our CIP-PIR model is that in (OO)-PIR the client is involved in the precomputation, whereas in CIP-PIR each server runs the precomputation locally without even knowing the identity of the client(s). Our protocol can be mapped into the (OO)-PIR model by moving the first message of our online phase into preprocessing and keeping a state of 128 bits for the seed and one block per query. Our client-independent preprocessing is substantially more powerful as it allows parallelization and amortization across *all* clients.

Two very recent multi-server PIR protocols [33, 44] in the OO-PIR model allow to efficiently retrieve a bit from the database with sublinear online complexity. These schemes are very efficient for retrieving small data but are inefficient when large values (like hashes in C3 applications, or files) need to be downloaded. In this case, Chor et al.-based PIR protocols like our CIP-PIR, or FSS-PIR [8] are better suited.

As the implementations of [13, 44] are not publicly available and [33] was parallel and independent work (which runs PIR on a single bit instead of larger messages), we leave an experimental comparison with these works to future work.

GPU-accelerated PIR. The first GPU-accelerated PIR scheme was shown by Melchor et al. [38, 39]. They utilize GPUs to improve the runtime efficiency of their lattice-based single-server PIR scheme by factor $\approx 10\times$. However, the server needs to compute many modular multiplications, so this scheme is still very inefficient. Mane et al. [36] replace the modular multiplications with vector additions on a GPU resulting in a much cheaper cost per bit ratio.

Marueac et al. [37] develop general techniques to improve single-server PIR schemes by using CUDA exemplarily on the PIR protocol of Kushilevitz and Ostrovsky [34]. This scheme requires large integer multiplications and modulo products among the whole database, which can be perfectly parallelized by GPUs. Another optimization introduces a preprocessing phase that takes place before the data is copied into the GPU's global memory. In the preprocessing phase, each block is padded such that the next sequence of blocks starts with a memory address that is a multiple of 16 bytes.

Dai et al. [14] use GPUs to improve Somewhat Homomorphic Encryption (SWHE)-based single-server PIR schemes [18]. They developed CUDA code that allows efficient modular multiplications and modulus switching, which is the main bottleneck of many single-server PIR protocols.

To the best of our knowledge, all previous works on using GPUs to accelerate PIR were for single-server PIR which is very compute intensive. A reason might be that multi-server PIR schemes rely on very cheap operations like XOR s.t. copying the relevant data into the GPU would eliminate the performance improvement. In this paper, we show for the first time in multi-server PIR how to precompute large parts of the server's answers independent of the client and thereby we can benefit from GPU acceleration here as well.

Acknowledgments. We thank our shepherd Marina Blanton and the anonymous USENIX Security'22 reviewers for helping us to improve our paper. This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by the Deutsche Forschungsgemeinschaft (DFG) — SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230, and by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within ATHENE. This work was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office, and by the Alter Family Foundation.

Availability. Our code is available under the MIT license at https://encrypto.de/code/cip-pir.

References

[1] Kinan Dak Albab, Rawane Issa, Mayank Varia, and Kalman Graffi. Batched differentially private information retrieval. *IACR Cryptology ePrint Archive, Report 2020/1596*, 2020. https://ia.cr/2020/1596.

[2] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *S&P*. IEEE, 2018.

[3] Vladimir L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of an oriented graph. *Journal of USSR Academy of Sciences*, 1970.

[4] Dmitri Asonov. *Querying databases privately: a new approach to private information retrieval*, volume 3128 of *LNCS*. Springer, 2004.

[5] Daniel Augot, Françoise Levy-Dit-Vehel, and Abdullatif Shikfa. A storage-efficient and robust private information retrieval scheme allowing few servers. In *CANS*. Springer, 2014.

[6] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *CRYPTO*. Springer, 2000.

[7] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *S&P*. IEEE, 2021.

[8] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*. Springer, 2015.

[9] Stefan Brechtken. GPU and CPU acceleration of a class of kinetic lattice group models. *Computers and Mathematics with Applications*, 2014.

[10] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*. IEEE, 1995.

[11] Catalin Cimpanu. Chrome 79 released with tab freezing, back-forward caching, and loads of security features. https://www.zdnet.com/article/chrome-79-released-with-tab-freezing-back-forward-caching-and-loads-of-security-features/, 2019.

[12] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*. USENIX Association, 2017.

[13] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT*. Springer, 2020.

[14] Wei Dai, Yarkin Doröz, and Berk Sunar. Accelerating SWHE based pirs using GPUs. In *FC*. Springer, 2015.

[15] Daniel Demmler, Amir Herzberg, and Thomas Schneider. RAID-PIR: Practical multi-server PIR. In *CCSW*. ACM, 2014.

[16] Daniel Demmler, Marco Holz, and Thomas Schneider. OnionPIR: Effective protection of sensitive metadata in online communication networks. In *ACNS*, 2017.

[17] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In *CCS*. ACM, 2013.

[18] Yarkin Doröz, Berk Sunar, and Ghaith Hammouri. Bandwidth efficient PIR from NTRU. In *FC*. Springer, 2014.

[19] Zeev Dvir and Sivakanth Gopi. 2 Server PIR with sub-polynomial communication. In *STOC*. ACM, 2015.

[20] ENZOIC. Detect compromised passwords. https://www.enzoic.com/, 2016.

[21] ENZOIC. LastPass selects PasswordPing for compromised credential screening. https://www.enzoic.com/lastpass-selects-passwordping-for-compromised-credential-screening/, 2016.

[22] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*. Springer, 2005.

[23] Craig Gentry and Shai Halevi. Compressible FHE with applications to pir. In *TCC*. Springer, 2019.

[24] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. In *STOC*. ACM, 1998.

[25] Ian Goldberg. Improving the robustness of private information retrieval. In *S&P*. IEEE, 2007.

[26] Andy Greenberg. Hackers are passing around a megaleak of 2.2 billion records. https://www.wired.com/story/collection-leak-usernames-passwords-billions/, 2019.

[27] Daniel Günther, Maurice Heymann, Benny Pinkas, and Thomas Schneider. GPU-accelerated PIR with Client-Independent Preprocessing for Large-Scale Applications. *IACR Cryptology ePrint Archive, Report 2021/823*, 2021. https://ia.cr/2021/823.

[28] Daniel Günther, Marco Holz, Benjamin Judkewitz, Helen Möllering, Benny Pinkas, and Thomas Schneider. PEM: Privacy-preserving epidemiological modeling. *IACR Cryptology ePrint Archive, Report 2020/1546*, 2020. https://ia.cr/2020/1546.

[29] Ryan Henry, Yizhou Huang, and Ian Goldberg. One (block) size fits all: PIR and SPIR with variable-length records via multi-block queries. In *NDSS*. The Internet Society, 2013.

[30] Troy Hunt. Have i been pwnd? https://haveibeenpwned.com/, 2019.

[31] Ari Juels and John G. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *NDSS*. The Internet Society, 1999.

[32] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*. ACM, 2020.

[33] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *USENIX Security*. USENIX Association, 2021.

[34] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*. IEEE, 1997.

[35] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *CCS*. ACM, 2019.

[36] Sunil B. Mane, Sandip B. Bansode, and Pradeep K. Sinha. Optimized private information retrieval using graphics processing unit with reduced accessibility. In *International IT Conference & Exhibition (CUBE)*. ACM, 2012.

[37] Mihai Maruseac, Gabriel Ghinita, Ming Ouyang, and Razvan Rughinis. Hardware acceleration of private information retrieval protocols using gpus. In *ASAP*. IEEE, 2015.

[38] Carlos Aguilar Melchor, Benoît Crespin, Philippe Gaborit, Vincent Jolivet, and Pierre Rousseau. High-speed private information retrieval computation on GPU. In *SECURWARE*. IEEE, 2008.

[39] Carlos Aguilar Melchor and Philippe Gaborit. A lattice-based computationally-efficient private information retrieval protocol. In *WEWORC*. Springer, 2007.

[40] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *STOC*. ACM, 1999.

[41] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security*. USENIX Association, 2015.

[42] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *USENIX Security*. USENIX Association, 2014.

[43] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen Mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPOPP*. ACM, 2008.

[44] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce M. Maggs. Puncturable pseudorandom sets and private information retrieval with polylogarithmic bandwidth and sublinear time. In *CRYPTO*. Springer, 2021.

[45] Jeff Shiner. Finding pwned passwords with 1password. https://blog.1password.com/finding-pwned-passwords-with-1password/, 2019.

[46] Sandeep Tamrakar, Jian Liu, Andrew Paverd, Jan-Erik Ekberg, Benny Pinkas, and N. Asokan. The circle game: Scalable private membership test using trusted hardware. In *ASIACCS*. ACM, 2017.

[47] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, and et al. Data breaches, phishing, or malware? Understanding the risks of stolen credentials. In *CCS*. ACM, 2017.

[48] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security*. USENIX Association, 2019.

A Summary of RAID-PIR [15, 16]

Our protocol is based on RAID-PIR [15,16] which we summarize next. RAID-PIR is an information-theoretic multi-server PIR scheme based on Chor et al.'s PIR [10]. These schemes use very efficient XOR operations and assume that less than $t \leq n$ of the n servers are colluding.

We first give an informal description of the scheme of Chor et al. [10]. In the two-server version, the input data D is split into a database DB of B blocks of size b each. If the client wants to learn block i, it sends to the first server a random B-bit string q_0, and sends to the second server a string q_1 which is equal to q_0, except that the i-th bit is flipped. Each server computes the XOR of the blocks which correspond to '1' bits in the string that it received, and sends the resulting b-bit block to the client. The client then computes the XOR of the two received blocks which is equal to the i-th block. The total communication with each of the n servers is $B+b$ bits.

In RAID-PIR [15, 16], the B blocks are split into n chunks and $t \leq n$ chunks are sent to each server, so each server stores t/n of the database. Consequently, the client's queries are shorter and each server only XORs a smaller subset of the blocks (cf. Fig. 6). As before, the XOR of all n queries is equal to a B-bit zero string with a '1' bit at the block that the client wishes to learn. A crucial observation that is used to improve performance is that for any specific block, out of the tk bits ($k = B/n$ is the number of blocks per chunk) that instruct t servers what to do with this block, $(t-1)k$ bits can be pseudo-random and only k bits need to be explicitly set to ensure that the result of the XOR is correct. Therefore, instead of sending a full length string to each server, the client can send to each server a seed that is used to compute a $(t-1)/n$ fraction of the string that the server must use. This cuts the communication from the client to the server by factor $t\times$.

q_0	011010	100010	011011	
q_1		101101	010110	100001
q_2	001101		001101	101001
q_3	010111	001011		001000
$q = e_9$	000000	000100	000000	000000

Figure 6: Example RAID-PIR queries with $n = 4$ servers, $B = 24$ blocks, $n = 4$ chunks, chunk size $k = B/n = 6$, and collusion threshold $t = 3$. The orange cells are the flip chunks while the white cells are the pseudo-random sub-queries. The client requests the block at index $i = 9$.

B Security Proof of CIP-PIR

Claim 1 *RAID-PIR (cf. §A) is multi-query secure according to Definition 3.*

Proof: Definition 3 requires that for any two equal-length sequences R, R' of queries, no subset of less than t servers can distinguish between the view it observes for queries R and R'.

Let us assume first that instead of depending on a PRG and using strings of the form $PRG(S_i)$ as the "non-flip chunks", the client only generates and sends truly random strings. In this information-theoretic version of the protocol, it holds for any block B_i that any t different shares of this block are uniformly distributed under the constraint that the exclusive-or of all t shares is equal to the client's query. Therefore, any subset of $t-1$ of these shares is uniformly distributed. The view of any coalition of $t-1$ servers can be fully simulated given their $t-1$ shares. Since these shares are uniformly distributed, this view is independent of the client's request, and is identically distributed for query sequences R and R'.

Consider now the RAID-PIR protocol, where the non-flip chunks are generated by a PRG. Suppose that there is a polynomial-time algorithm D which can distinguish between the view of the coalition of the $t-1$ servers for two sequences of requests of equal length, R and R'. This algorithm D is not able to distinguish between these two views in the information-theoretic version of the protocol. The difference between the two views is whether the string of the t-th server that is used to retrieve an item(s) (the server which is not controlled by the attacker) is pseudo-random or random. Therefore, D could contradict the assumption that the PRG is secure, by distinguishing between the outputs of the PRG (for which it succeeds in distinguishing the two views) and truly uniform strings (for which it does not).

Claim 2 *For semi-honest servers, CIP-PIR (cf. §3.2) is multi-query secure according to Definition 3.*

Proof: For semi-honest servers the only difference between RAID-PIR and CIP-PIR is that in the latter protocol the non-flip chunks of the corrupt $t-1$ servers are chosen by the servers, rather than by the client. Let us prove that any corrupt subset of $t-1$ servers cannot distinguish between its views in any two request sequences R and R'.

Consider two settings: In the U-setting all servers outside the corrupt subset use uniformly random non-flip chunks. In the PRG-setting they generate these chunks using a PRG, as is defined by the protocol.

For some parts of the database, all the servers in the corrupt subset obtain non-flip chunks. These chunks are obviously independent of the client requests R, R' since they were generated by the servers themselves. For all other parts of the database, exactly one of the servers in the corrupt subset obtains a flip chunk sent by the client. The value of this flip chunk is equal to the exclusive-or of the query, the non-flip chunks chosen by the other corrupt servers, and, most importantly, at least one non-flip chunk chosen by a (non-corrupt) server which is not a member of the corrupt subset. In the U-setting the non-flip chunks outside the corrupt subset are uniformly random and therefore the view of the corrupt subset is also uniformly random and D cannot distinguish between the views for R and R'. Assume that in the PRG-setting the corrupt subset can run a polynomial-time algorithm D which can distinguish between its views for request sequences R and R'. The existence of an algorithm D which distinguishes between R and R' in the PRG setting can be used to contradict the assumption that the PRG is secure. This can be shown by a standard hybrid argument: The first hybrid H_0 is identical to the U-setting where all non-corrupt servers use uniformly random non-flip chunks. In H_0, D cannot distinguish between R and R' better than guessing the result. Hybrid H_i is where the first i non-corrupt servers use a PRG and the remaining $n-(t-1)-i$ non-corrupt servers use uniformly random strings. Hybrid H_{n-t+1} is identical to the PRG-setting. In H_{n-t+1}, D can distinguish between R and R' with a non-negligible advantage over guessing. Therefore there exists an $1 \leq i \leq n-(t-1)$, for which the difference between the success probability of D when working in H_{i-1} and in H_i is non-negligible. In other words, D can be used to distinguish between the case that a string is the output of the PRG or uniformly random, if we plug that string as the values used by the i-th server in H_i. This contradicts the security of the PRG.

Claim 3 *For malicious servers, CIP-PIR (cf. §3.2) is multi-query secure according to Definition 3.*

Proof: Note that the security definition of PIR (as in Definition 3) is only concerned with the privacy of the client, and not with the correctness of the protocol. Namely, the definition requires that corrupt servers cannot distinguish between two different query sequences of the clients. In the protocol, a server sends a seed S_i to the client, receives a query from the client, and sends an answer. This is done independently for each query. Therefore, the only operation of malicious servers that can affect the information that they receive from the client (and can therefore lead to them breaking Definition 3) is changing the seeds that they send to the client, with the goal that this change will result in the client sending information that enables breaking Definition 3. (For example by resending the same seeds.) Recall that for a query of the client, the client sends information that is embedded in a flip chunk that is sent to one server. This information is based on the client input and on the seeds received from $t-1$ other servers. The flip chunk is computed as the exclusive-or of the expansion of the $t-1$ seeds and the query.

There are only two possible cases: In the first case one of the corrupt servers is the recipient of the flip chunk. In the second case, the recipient of the flip chunk is not corrupt.

In the first case, it must hold that at least one of the $t-1$ seeds that were expanded to strings that were XORed into the flip chunk, was generated by an honest server. This seed is random and unknown to the corrupt servers, and therefore the string that is XORed into the flip chunk looks pseudo-random to them. Therefore, a standard argument can show that if they can distinguish that flip chunk from a random string then they can also break the pseudo-randomness of the pseudo-random generator that was used to expand the seed.

The other case is where all $t-1$ non-flip chunks affecting the generation of a flip chunk are chosen by corrupt servers. In this case the resulting flip-chunk is sent to another server, which is not part of the corrupt coalition, and therefore the security property required by Definition 3 is preserved. (We must comment that in this case the corrupt servers might cause the flip chunk to reveal information about the queries. For example, if they repeat using the same seeds for two queries, the exclusive-or of the flip-chunks of the two queries will be equal to the exclusive-or of the queries. But since these flip chunks are sent to an additional server which is not part of the coalition, the requirement of Definition 3 is preserved.)[7]

C Complexity Analysis of CIP-PIR

In this section we compare the communication, computation and storage complexities of RAID-PIR [15, 16] and our new CIP-PIR scheme (cf. §3.2). We further show experimental delay times and storage overheads of CIP-PIR.

Complexities. Table 2 compares the communication, computation and storage complexities of RAID-PIR and CIP-PIR. To minimize the number of variables, we set the blocksize $b = \sqrt{|DB|}/n$ which is the optimal blocksize for n servers and database size $|DB|$ (cf. §3.3) . The number of blocks is $B = |DB|/b = n\sqrt{|DB|}$ and the number of blocks per chunk is $k = B/n = \sqrt{|DB|}$.

Communication. The total amount of communication is the same in both schemes. In both schemes, a κ bit seed is uploaded (RAID-PIR) or downloaded (CIP-PIR). The query for both schemes has $B/n = \sqrt{|DB|}$ bits and an answer from one server has size $b = \sqrt{|DB|}/n$, i.e., all n answers have in total size $\sqrt{|DB|}$. However, our CIP-PIR scheme needs one additional round-trip to receive the seeds from the servers, which results in slightly higher communication time.

Server Computation. The server's average online computation in our CIP-PIR protocol is $r\times$ smaller than in RAID-PIR. In CIP-PIR, one server processes only one chunk of size $kb = |DB|/n$ whereas a RAID-PIR server processes r chunks, where r is the threshold and k is the number of blocks per chunk. We give the average number of XOR operations as the actual number depends on the number of 1-bits in the client's query that is on average $k/2$ per chunk. Thus, we assume that a server only needs to touch $k/2$ blocks per chunk. Note that the database preprocessing (cf. §2.1) that is used by RAID-PIR and in our implementation improves the costly dependence on the client's query to a constant number of XOR operations (cf. [16] for details). This is done by building groups of, e.g., eight blocks, precomputing all 2^8 linear combinations of the corresponding sub-query, and XOR only one

[7]In order to prevent even this attack, the redundancy parameter/threshold t can be set under the assumption that at most $t-2$ servers collude, and therefore all flip-chunks depend on at least one legitimate seed.

Scheme	Communication	RTT	Server Computation (avg.)	Client Computation	Storage
RAID-PIR [16]	$n(2\sqrt{\lvert DB\rvert/8n}+\kappa/8)$	1	**Online:** $r\lvert DB\rvert/(2n)$	$\sqrt{\lvert DB\rvert}(rn+1+1/n)$	$\lvert DB\rvert r/n$
CIP-PIR [this work]	$n(2\sqrt{\lvert DB\rvert/8n}+\kappa/8)$	2	**Offline:** $(r-1)\lvert DB\rvert/(2n)$	$\sqrt{\lvert DB\rvert}(rn+1+1/n)$	$\lvert DB\rvert r/n+\lvert Q\rvert(\sqrt{\lvert DB\rvert}/n+\kappa)$
			Online: $\lvert DB\rvert/(2n)$		

Table 2: Comparison of communication, number of round trips (RTT), number of XOR operations for one server and for the client, and storage per server for RAID-PIR [15, 16] and our CIP-PIR protocol (§3.2) with n servers holding a database of size $|DB|$ with threshold r and symmetric security parameter κ. The computation is based on the optimal blocksize $b=\sqrt{|DB|}/n$ (cf. §3.3). The preprocessing queue of our CIP-PIR protocol has $|Q|$ entries.

DB Size (GB)	Queue Size (MB)	Offline Computation (s)	Simultaneous Queries	Delay avg. (ms)
0.8	142	214	1	23
			10	96
			100	1 091
4	316	1 003	1	93
			10	364
			100	4 619
8	447	1 996	1	176
			10	737
			100	8 500

Table 3: Queue sizes, offline computation times, and avg. delays until the client receives the desired block of our CIP-PIR protocol (cf. §3.2)with $n=2$ servers. The offline computation is the total time for filling the empty preprocessing queue with $|Q|=10\,000$ entries.

block per group depending on the query. The number of XOR operations gets smaller with increasing threshold r as $r-1$ chunks are processed in the offline phase.

Client Computation. The client computation complexity is equal for both schemes. A client XORs r times a bit per block, which are in total $Br=rn\sqrt{|DB|}$ XOR operations. After the client receives all blocks from the servers, she XORs all of them to compute the requested block, which are in total $(n-1)b=\sqrt{|DB|}(1+1/n)$ XOR operations.

Storage. Finally, a (CIP-)RAID-PIR server needs to store r/n of the database, while the CIP-PIR server additionally stores $|Q|$ (seed, value)-pairs of size $b+\kappa=\sqrt{|DB|/n}+\kappa$. Setting $\kappa=128$ bit and $|Q|\ll B$, the storage overhead is negligible compared to the performance gain of CIP-PIR. Concretely, the queue size for the (seed, value)-pairs is equal to the database size if $|Q|\approx\sqrt{|DB|}$.

Storage and Delays. In Table 3, we show the queue sizes, the offline computation time, as well as the min., max., and avg. delays of CIP-PIR. We use three clients who in parallel flood the CIP-PIR servers with 1, 10, and 100 queries to simulate simultaneous queries. The min./max./avg. delay is the smallest/highest/average time a client has to wait until she obtains the desired PIR block. The offline computation time is the total time for filling the server's empty preprocessing queue Q with $|Q|=10\,000$ entries. The total storage is the sum of the database size $|DB|$ and the queue size $|Q|$.

As already observed in Table 2, the queue size grows sublinearly with the database size, which we can also observe in Table 3. While the difference between the queue size of a 0.8 GB and 4 GB (5× larger) database is 174 MB, the difference between the 4 GB and 8 GB (only 2× larger) database is just 125 MB. For the largest database of 8 GB, the queue size $|Q|=89\,427$ is equal to the database size.

The offline computation time grows linearly with the database size (cf. Table 2), which we can approximately also see in Table 3. A CIP-PIR server needs $\approx$ 34 minutes to precompute 10 000 pairs (200 ms per pair) in the offline computation for the largest database of 8 GB.

Our CIP-PIR implementation processes incoming queries sequentially in a "first-come first-serve" manner. Thus, the delay time until a client obtains a block highly depends on the number of simultaneous queries as shown in Table 3. For the 8 GB database, the delay for a single query is just 176 ms, but for 10 simultaneous queries the average delay time is 737 ms and for 100 queries it is 8 500 ms. Hence, the performance of our PIR scheme depends on the database size and the number of active users. Note that our servers just use the computation power of one machine. Thomas et al. [48] deploy their GPC tool with Google Cloud Functions, which scales with the number of incoming queries. Integrating our protocol in their system or optimizing our implementation for hardware-based parallelization would yield better average delay times.