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 primitive that allows a client to securely query a database entry from $n \geq 2$ non-colluding servers, which 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 $\frac{n-1}{n}$ of the online computation to a local preprocessing phase suitable for efficient batch precomputations. The security and online performance of CIP-PIR improve 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 and further accelerating the huge amount of XOR operations with GPUs. 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 security and online performance simultaneously increase with the number of servers n. In addition, CIP-PIR is the first multi-server PIR scheme that is accelerated by GPUs. It achieves an improvement up to factor 2.1× over our CPU-based implementation. Moreover, a client can access a database entry of a 25 GByte 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) [49], private blocklists in webbrowsers [34], and COVID-19 contact tracing 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. This leakage allows certain attacks [36].

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 several non-colluding servers. We discuss in §1.3 the applicability of this setting to C3, as well as to other applications. To further improve the amortized runtime, we use GPUs to accelerate (a) the offline preprocessing by batching multiple queries completely independent of the client, and (b) large parts of the online computations.

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. 7). A C3 application has billions of entries and requires the user to wait for the response, so even multiple seconds are not tolerable. Hence, we build a new multi-server PIR protocol called CIP-PIR. As a basis, we use the RAID-PIR protocol [16, 17], which is a multi-server PIR scheme with $n \geq 2$ non-colluding servers that extends Chor et al.'s PIR scheme [11]. 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 $\frac{n-1}{n}$ 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 $t < n$ servers may collude), the performance of the online phase of our CIP-PIR scheme becomes *independent* of the corruption 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 Private Information Retrieval for multiple applications. Our CIP-PIR construction can be used in any PIR application which requires low latency for large-scale data. Two important and omnipresent applications are in the context of compromised credential checking (C3) and the COVID-19 pandemic described below. Further applications with these requirements are, for example, private queries to medical and patent databases [4] and anonymous messaging [17]. 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 [34]. 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, 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 [27]. Thomas et al. [48] 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 [36] that allow users to check whether their credentials are breached or not.

Popular password managers already integrate C3 services: 1password uses *HaveIBeenPwnd* (HIBP) [30, 46] and LastPass uses ENZOIC [21, 22]. 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. [49] 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. [49] for further discussions). Recently, Thomas et al. [49] 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 [36]. 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. [36] 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. [49] 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 [12]. 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. [49] and Li et al. [36] 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, [49] and [36] 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.

COVID-19 Contact Tracing. Another currently relevant example related to the COVID-19 pandemic is in the context of contact tracing: instead of publishing the BLE ephemeral IDs sent by COVID-19 positive persons, the application can let users run a PIR protocol to query if they received any message which appears in a dataset of messages sent by people who were found to be COVID-19 positive. This application demonstrates a potential need for very large-scale PIR, where different entities are likely to be willing to collaborate in order to support PIR. Further applications of PIR related to the COVID-19 pandemic might be for hotspot detection [6] and epidemiological modeling [28].

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. [36], 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, Mozilla's Firefox will use the Prio system for gathering telemetry from browsers [13]. The second party in this case will be ISRG – the Internet Security Research Group, which also runs the Let's Encrypt certificate authority, and is therefore an entity which is trusted by Internet users and is a separate than Mozilla.[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 a C3 feature for the Safari browser, that "doesn't reveal your password information – even to Apple" (https://www.apple.com/ios/ios-14/). Microsoft introduced a similar feature for Edge in 2021 (https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge). 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 will 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 GBytes large databases and outperforms recent efficient PIR implementations like PIR based on function secret sharing [8]. 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 (§3.2). 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 [33]). Today, this model is the state-of-the-art for all efficient MPC protocols. In concurrent and independent work to our paper, the preprocessing model was applied to multi-server PIR in [34] where the server interacts with a specific 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 for the first time *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 (§3.3). We propose the first PIR scheme in our new CIP PIR model called CIP-PIR. Our protocol is based on the very simple RAID-PIR scheme by Demmler et al. [16, 17] and moves a part of the client's query generation to the server sides. This costs and additional round trip, but allows for very efficient offline preprocessing without involving the client. Moreover, the preprocessing phase can

[1] See, for example https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/, and the discussion in slide 132 of https://rwc.iacr.org/2020/slides/Gibbs.pdf on finding a partner for running the second server.

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. [11]) has linear communication complexity, but this is only one bit per block. In §B we show that even for a 16 TByte database and n=2 servers, our concrete communication is only 4 MBytes and hence only 2x larger than today's most communication-efficient schemes with sub-linear communication complexity [8].

Database Compression in PIR (§3.4). 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 [49] 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 PIR schemes, e.g., the linear XOR operations of PIR based on function secret sharing [8, 9] 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) than RAID-PIR [17] 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 [10, 44]. 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 $n = 3$ 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 in [11], a client wishes to learn one or multiple blocks from a public database $|DB|$ held by n servers without revealing information about the query. We focus on *multi-server PIR* schemes since they have lower computational overhead for the servers and the (pontentially mobile) client. In multi-server PIR the database DB is split over $n \geq 2$ servers that are assumed to not collude and the client sends a request to each server.

PIR Model. We define a classical PIR protocol as a tuple of algorithms (*Create*, *Request*, *Response*, *Combine*) as summarized in Fig. 1 and described below:

Create is locally run once by the owner of the database and takes as input some data D and outputs the database DB_i for each server $i \in [n]$. (Note that unlike the original PIR constructions of [11], each server might have a different state). A database DB is generated from the data D and a unique part of the database denoted as DB_i is sent to server i.

Request is run by the client and takes as input the index idx of the data to access and outputs a list of queries $(q_0, \dots, q_{n-1})$, 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 $a_0, \dots, a_{n-1}$ from the servers and calls the *Combine* algorithm that outputs the desired data $d = D[idx]$.

Definition 1 (Multi-query security of PIR) *A PIR scheme with redundancy parameter/threshold $1 < t \leq n$ is called* secure *if any coalition of less than t servers does not learn any information about the query indexes. Namely, for any data set D, and any two sequences of (possibly not all unique) requests $R = (idx_1, \dots, idx_m)$ and $R' = (idx'_1, \dots, idx'_m)$, no polynomial time algorithm can distinguish the view of the servers in the coalition (consisting of the requests they receive and the messages they send) when receiving the requests in R, and their view when receiving R'.*[2]

The original PIR protocol of Chor et al. [11] 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 is called *correct* if the client always recovers the correct output, namely $D[idx] = Combine(Response^0(Request(idx)[0]), \dots, Response^{n-1}(Request(idx)[n-1]))$.

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

[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" [25]. 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 [23, 41].

Server $i \in [n]$		**Client**
Setup Phase (once for all servers):		
input: data D		
$DB_i \leftarrow Create(D)$		
Online Phase:		**input**: idx
		$(q_0, \ldots, q_{n-1}) \leftarrow Request(idx)$
	$\xleftarrow{\quad q_i \quad}$	
$a_i \leftarrow Response(q_i, DB_i)$		
	$\xrightarrow{\quad a_i \quad}$	
		$d \leftarrow Combine(a_0, \ldots, a_{n-1})$
		output: $d = D[idx]$

Figure 1: Message flow for a classical PIR protocol. The client communicates with all n servers $i \in [n]$ in parallel.

2.2 CUDA

Nvidia's Compute Unified Device Architecture (CUDA) [10, 44] 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 blocks where each block has 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.

2.3 Threat Model

Throughout this work, we evaluate PIR protocols relative to following main threat model: a coalition of honest-but-curious PIR servers try to learn information about the client's query. In this threat model, servers follow the protocol correctly while trying to learn details about the PIR query. This is obliviously 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 threshold which should obliviously 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 summarize the RAID-PIR scheme from [16, 17] (§3.1), introduce our new Client-Independent Preprocessing PIR model (§3.2), describe our CIP-PIR scheme (§3.3), and present our new PIR database compression technique (§3.4). We give the security proof of CIP-PIR in §A, its complexity analysis in §B, and details of database updates in §C.

3.1 RAID-PIR [16, 17]

Our protocol is based on RAID-PIR [16, 17] which we summarize next. RAID-PIR is an information-theoretic multi-server PIR scheme based on Chor et al.'s PIR [11]. These schemes use very efficient XOR operations and assume that a subset of the n servers are non-colluding.

Let us first give an informal description of the scheme. In the two-server version of Chor et al.'s PIR [11], 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.

In RAID-PIR [16, 17] 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. 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

q_0	011010	100010	011011	
q_1		101101	010110	100001
q_2	001101		001101	101001
q_3	010111	001011		001000
e	000000	000100	000000	000000

Figure 2: Example RAID-PIR queries with $n = 4$ servers, $B = 24$ blocks, $n = 4$ chunks, chunk size $k = B/n = 6$, and 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 $idx = 9$.

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 an $\frac{r-1}{n}$ fraction of the string that the server must use. This cuts the communication from the client to the server by factor $t\times$. 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.

In more detail, RAID-PIR instantiates the four algorithms for our PIR model from §2.1 as follows:

Create. In the setup phase, the input data D is split into B blocks, each of b Bits. These blocks are grouped into n chunks. The resulting database DB is split over n servers. The threshold r with $2 \leq t \leq n$ specifies the number of servers that must be corrupted to break the PIR scheme, and thus also the number of chunks that each server has to process per query. Each server therefore receives only r chunks, which it needs for processing queries.

Request. The client calls the *Request* method on input idx to retrieve the block $d = D[idx]$. An example for generating the queries $q_0, \ldots, q_3$ for $n = 4$ servers is given in Fig. 2. For this, the client generates the main query e, which is a B-Bit zero vector with a 1 at position $idx = 9$. This string is secret-shared among all servers utilizing an XOR-based sharing. This sharing is generated by sampling a κ-Bit random seed S_i for each server and using a PRG to stretch it into $r-1$ pseudo-random so-called "non-flip chunks" each of size size $k = B/n$ Bits. Each server has a unique "flip chunk" that cancels out all unwanted Bits from the corresponding non-flip chunks of the other servers s.t. the XOR of all n queries results in the client's main query e. The client's query q_i sent to server i then consists of the k-Bit flip chunk $flip_i$ and the seed S_i.

Response. When the server i receives a query, it calls the *Response* algorithm with the query $q_i = (flip_i, S_i)$ and the database DB_i as inputs. Having the seed S_i, the server computes its query $q = flip_i || PRG(S_i)$. Then, the server XORs all blocks which correspond to 1 Bits in the query q. As shown in [17], this step can be optimized by grouping some blocks and precomputing all possible block combinations utilizing the method of four Russians [3]. The resulting block a_i is sent back to the client.

Combine. The client calls the *Combine* method that computes $d = \bigoplus_{i=0}^{n-1} a_i$ to obtain $d = D[idx]$.

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

Previous works on PIR, such as [7, 17], 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 - the *database preprocessing* and the *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 [7, 17]. 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 [14] as it computes the preprocessing/online phase *without* involving the client.

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. 3.

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

Each server 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.

The *Request* algorithm takes as input the index idx of the data item to access, and seeds $S_1,\ldots,S_n$ obtained from the n servers, and generates queries $q_1,\ldots,q_n$. For each query, each server i pops one pair (S_i,A_i) from Q_i.

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.

3.3 Our CIP-PIR Protocol

RAID-PIR [16, 17] improves over Chor et al.'s scheme [11] 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 $2 \le r \le n$ 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 to compute in advance $(t-1)/n$ of the XOR operations, and only the remaining $1/n$ XOR operations are computed in the online phase, i.e., the online computation is independent of the threshold t. Then, the servers give the seed to the client and the protocol proceeds as before. Aside from the significant performance improvement, the security of the scheme grows with the threshold t, since the protocol allows up to $t-1$ servers to collude. Since the online computation is independent of the threshold t, the server's online performance is not affected by setting $t=n$ to achieve the highest security level. These two advantages are achieved since each server only has to process $1/n$-th of the database in the online phase. The remaining part of the database is touched in the parallel *Preprocess* algorithm introduced in §3.2.

Algorithm 1 *Create* of CIP-PIR

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

***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. 3, the server permanently computes (seed, value)-pairs (S,A) and pushes them to its local queue Q_i. The seed S is expanded to 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, ..., chunk_{t-1}) ← DB_i , k ← B/n
DB ← chunk_1 | ... | chunk_{n-1}                  ▷ all but first chunk
while Q_i is not full do
    S ←$ {0,1}^κ
    q ← PRG(S_i, k(t-1))                          ▷ non-flip chunk
    A ← q · DB                 ▷ XOR blocks of DB corresponding to q
    Q_i.push(S, A)                   ▷ push seed/value pair (S, A)
end while
return a_i                                        ▷ answer of server i
```

***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 vector to 1. Then, the client expands the seeds S_i to a $k(n-1)$ Bit sub-query v which covers all non-flip chunks of server i. Server i has u chunks to the left and w chunks to the right of its flip chunk (e.g., in Fig. 2, the server with the query q_2 has $u = w = 1$ chunk left and right of its orange flip chunk). 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 3 *Request* of CIP-PIR

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

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

Figure 3: Message flow for client-independent preprocessing PIR (CIP-PIR). The client communicates with all n servers $i \in [n]$ in parallel.

Algorithm 4 *Response* of CIP-PIR

input: q_i, DB_i, A_i ▷ query q_i, database DB_i, precomputed block A_i
$(chunk_0, \dots, chunk_{t-1}) \leftarrow DB_i$ ▷ $chunk_0$ 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: $a_0, \dots, a_{n-1}$ ▷ answers of the servers
$d \leftarrow \bigoplus_{i=0}^{n-1} a_i$
return d ▷ data block d

***Combine* (Algorithm 5).** As in RAID-PIR, 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]

[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].

3.4 PIR with Database Compression

In the following, we show how to compress the database in PIR schemes by (a) adapting a database compression technique which was used in [47] 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 [47] 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 re-

[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.

trieved block to decompress the block. For a better compression, we increase the blocksize b. Thus, we use less 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.

Shorter Hashes. In C3 [49], the compromised data is represented as a 32 Bytes hash prefix, which results in a database of $33 \cdot |DB|$ Bytes. Since we only need to check for equality, we can apply a trick from the PSI literature [18, 42, 43] and only use the first $40 + \log_2 |DB|$ Bits of $\mathcal{H}(H^b)$ 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 is an improvement 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 $5\times$ improvement factor. 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 our CIP-PIR scheme is

$$\mathcal{C}(b) = 1/8 \frac{|DB|}{bn} + \kappa/8 + b, \qquad (1)$$

where the first element 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 $\hat{b} = \sqrt{|DB|/8n}$. 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. 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 CIP-PIR

In CIP-PIR (cf. §3.3), the n servers precompute (seed, value)-pairs in a separate thread. Since these precomputations contain a huge amount of independent XOR operations, GPUs that are built for highly parallel data processing are a natural choice. The precomputations can be outsourced on GPU clusters to support PIR applications on a large-scale. In addition, the online phase can naturally be accelerated as well by utilizing the GPU for computing the *Response* algorithm as it works similarly.

In §4.1, we present two approaches for accelerating the huge amount of XOR operations of CIP-PIR with a GPU. In §4.2, we demonstrate how GPUs can substantially improve the amortized runtime by batching multiple queries.

4.1 GPU-Accelerated Tuple Computation

The massive number of XOR operations are the main cost factor of our CIP-PIR protocol. In this section, we demonstrate two approaches for parallelizing these computations efficiently on a GPU using Nvidia's CUDA architecture (cf. §2.2).

4.1.1 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, 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.

4.1.2 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.

4.2 Amortized Query Preprocessing

Batching multiple queries was already used in computational PIR schemes [2, 24], but required waiting for multiple client queries in IT-PIR schemes [1, 14] to collect several client requests which increases online latency. As the preprocessing phase of our CIP-PIR scheme is now completely independent

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

of the client, we can now for the first time batch multiple queries without increasing online latency.

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. §2.2). 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 is factor $0.7\times$ slower than in RAID-PIR for CPU and up to $53\times$ for GPU (cf. §5.2.1 for details).

5 Implementation and Benchmarks

We implemented a CPU-based (cf. §3.3) and a GPU-accelerated version (cf. §4) of our CIP-PIR protocol in C++. 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. §3.4).

5.1 Implementation

Our implementation consists of hree 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 pseudo-random values. This simulates the real-world deployment of C3 and COVID-19 related applications (cf. §1.2), as hashed values are pseudo-random as well and hence have the same distribution. The database is stored in the RAM of the OS and the GPU memory of the server. We created databases up to 3.5 billion entries which suffices to cover the password breaches in Collection 1-5 [27].

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 [17] 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 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 amoritzed runtimes of the preprocessing phase.

Influence of blocksize and number of pairs. In this benchmark, depicted in Fig. 4, 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 1 Mio entries of 8 Bytes and $n = 3$ servers, i.e., $2/3$ of the whole database is processed in the preprocessing phase. 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 $\approx 1.3\times$ is only measurable until all threads are occupied, but afterwards, the performance falls back to a factor of $\approx 1.1\times$ improvement over the non-batched execution. However, the CPU-based

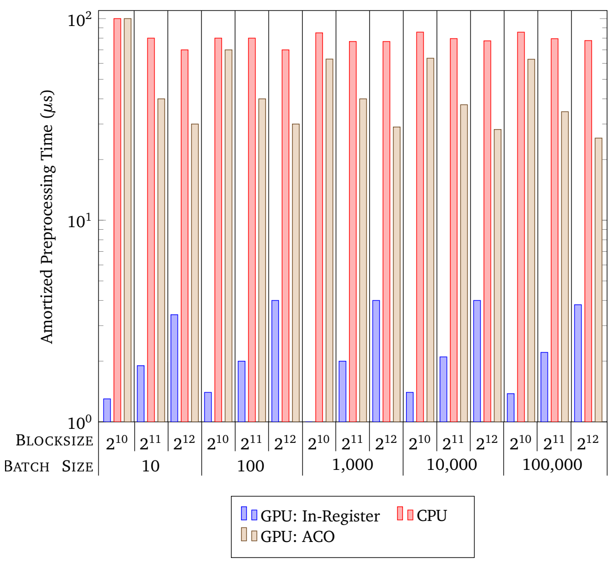

Figure 4: The amortized preprocessing time per (seed, value)-pair of our CPU-based and two GPU-accelerated implementations. We use $n = 3$ servers, a database of 1 Mio entries and an entry size of 8 Bytes. The blocksize is given in Bytes.

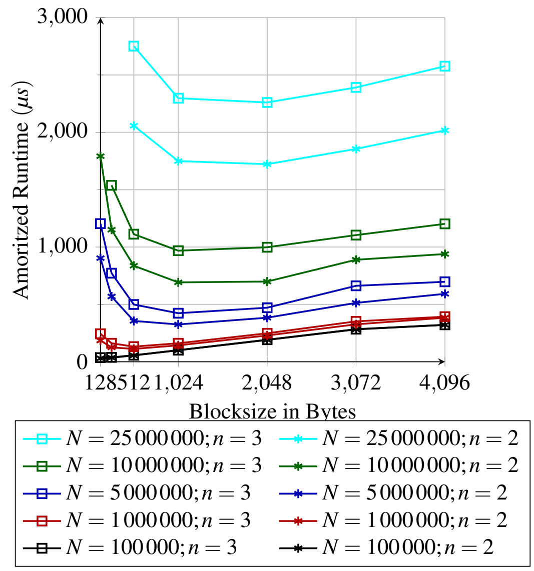

Figure 5: Runtimes for various blocksizes on the GPU-accelerated implementation using the in-register approach. Different colors indicate the number of entries in the database, whereby the marks show the amount of servers ($n = 2$ and $n = 3$) used in the corresponding experiment.

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. §3.4).

ACO: In the ACO parallelization technique (cf. §4.1.1), 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 good 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 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. §4.1.2), 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, depicted in Fig. 5, we measure the preprocessing runtime of the in-register-based implementation for various blocksizes, database sizes, as well as $n = 2$ and $n = 3$ servers to investigate whether an optimal blocksize for this approach exists.

We see in Fig. 5 that each database - except for the smallest one with only $100\,000$ 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 $N = 1$ Mio entries the optimal blocksize is 512 Bytes whereas with $N = 5$ Mio entries it is 1 KByte.

5.2.2 Setup and Online Phase

We compare our CPU-based CIP-PIR implementations (cf. §5.1) in C++, our reimplementation of the RAID-PIR protocol [16, 17] using the same codebase (including the parallelization and pipeline optimizations outlined in §5.1), the original Python implementation of RAID-PIR from [16, 17] 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 and CIP-PIR. 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 §3.4, the blocksize is reduced by factor $\approx 1.2\times$ to $b \approx 73$ KB, which perfectly matches with the theoretical analysis.

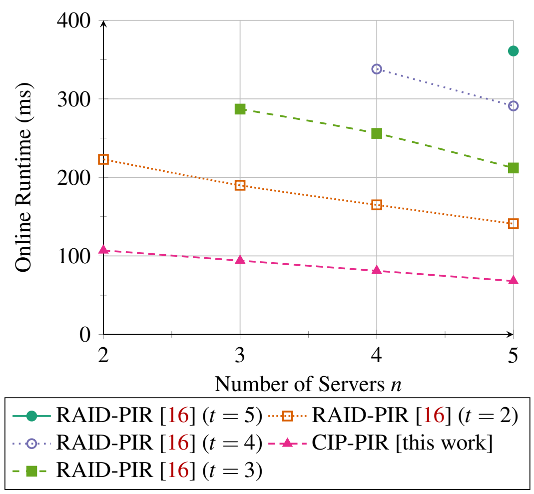

Figure 6: Online runtimes of our CPU-based PIR implementations for different number of servers n on a 500 MB database. Our implementation of the RAID-PIR protocols [16, 17] uses the same codebase as CIP-PIR. The threshold t of CIP-PIR is set to n.

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 $2 \leq t \leq n$ 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. 6, where 2 to 5 servers operate on a 500 MB database and achieve improvements of $\approx t\times$.

Note that Fig. 6 shows all possible combinations of number of servers n and threshold t for RAID-PIR, but not for CIP-PIR as its online computation is independent of t (cf. §B).

We see that the runtime of querying one entry decreases with the number of PIR servers n. At some point, however, the client's overhead of managing multiple connections becomes high and the improvement of deploying many servers is reduced (e.g., the improvement of CIP-PIR from 2 to 3 servers is higher than from 4 to 5 servers). Each service provider profits from many deployed servers n as the server only touches $1/n$ of the DB in the online phase. The client, however, has more work to do since she needs to handle n connections and needs to perform more XOR operations and PRG evaluations for building the queries.

[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.

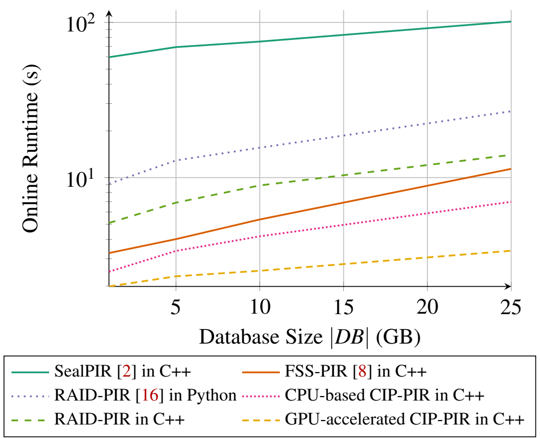

Figure 7: Online runtimes of PIR implementations for different database sizes $|DB|$ and $n = 22$ servers ($n = 1$ for single-server SealPIR [2]). The threshold for all RAID-PIR implementations is $t = 2$.

Communication. CIP-PIR and RAID-PIR have the same amount of communication (independent of RAID-PIR's 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. The amount of upload and download data is almost equal, but due to our compression from §3.4, we are not able to choose the optimal blocksize $\hat{b}$ for our database.

Comparison with other PIR implementations. In Fig. 7 we compare the online runtimes of several PIR implementations for varying database sizes. We compare the communication complexity of several PIR schemes in §B. For

RAID-PIR [16, 17] we set the 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. §4.1.2). 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 [16, 17] 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. 7. 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.

Although FSS-PIR [8] is well known for its very efficient logarithmic upload communication complexity, our CPU-based CIP-PIR implementation improves over the state-of-the-art implementation of FSS-PIR [32] by factor $\approx 1.6\times$ in runtime ($\approx 3.5\times$ with our GPU-based implementation) for $|DB| = 25$ GByte. In §B, we show that the communication complexity of CIP-PIR for retrieving a $b = 1$ MByte block even from a $|DB| = 16$ TByte database the communication is only 4 MByte and thus only $2\times$ more than FSS-PIR [8] and hence the effect of network bandwidth is negligible.

6 Related Work

Multi-Server PIR. Chor et al. [11] 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 [5,20,26,29] 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. [8, 9]. 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. [11]-based PIR for large databases, which we significantly improve in our work.

Previous works [7, 16, 17] 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 concurrent and independent work to our paper, The authors of [14] introduced 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 she wants to access. The difference between their model and our new CIP-PIR model is that in our protocol, the server does the precomputations locally without even knowing the client(s). Our protocol can be mapped into their PIR model by moving the first message of our online phase into the preprocessing phase and keeping a state of only 128 Bit 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 [34, 45] 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 become inefficient when large values (like truncated hashes in our C3 application) or even files need to be downloaded. In this case, Chor et al. [11]-based PIR protocols like our CIP-PIR, or FSS-based PIR [9] are better suited.

GPU-accelerated PIR. The first GPU-accelerated PIR scheme was shown by Melchor et al. [39, 40]. 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. [37] replace the modular multiplications with vector additions on a GPU resulting in a much cheaper cost per Bit ratio.

Marueac et al. [38] develop general techniques to improve single-server PIR schemes by using CUDA exemplarily on the PIR protocol of Kushilevitz and Ostrovsky [35]. 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. [15] use GPUs to improve Somewhat Homomorphic Encryption (SWHE)-based single-server PIR schemes [19]. 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

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.

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 *Lecture Notes in Computer Science*. 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] Alexandros Bampoulidis, Alessandro Bruni, Lukas Helminger, Daniel Kales, Christian Rechberger, and Roman Walch. Privately connecting mobility to infectious diseases via applied cryptography. *IACR Cryptology ePrint Archive, Report 2020/522*, 2020.

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

[8] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. IEEE, 2021.

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

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

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

[12] 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.

[13] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2017.

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

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

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

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

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

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

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

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

[22] ENZOIC. LastPass selects Password-Ping for compromised credential screening. https://www.enzoic.com/lastpass-selects-

passwordping-for-compromised-credential-screening/, 2016.

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

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

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

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

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

[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] Daniel Kales. C++ dpf library. https://github.com/dkales/dpf-cpp, 2019.

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

[34] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. *IACR Cryptology ePrint Archive, Report 2021/345*, 2021. To be published at Usenix Security 2021. https://ia.cr/2021/345.

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

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

[37] 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.

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

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

[40] Carlos Aguilar Melchor and Philippe Gaborit. A lattice-based computationally-efficient private information retrieval protocol. In *Western European Workshop on Research in Cryptology (WEWORC)*. Springer, 2007.

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

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

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

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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 Security Proof of CIP-PIR

Claim 1 *RAID-PIR (cf. §3.1) is multi-query secure according to Defintion 1 on page 4.*

Proof: 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 the t different shares of this block, which consist of $t-1$ shares in non-flip chunks and one share in a flip chunk,are all 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. Consider any coalition of $t-1$ servers. The view of theseservers can be fully simulated given their $t-1$ uniformly distributed shares and is therefore independent of the client's request.

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. Therefore, D could be used to distinguish between outputs of the PRG (for which it succeeds in distinguishing the two views) and truly uniform strings (for which it does not), and thus break the security of the PRG, contradicting the assumption that the PRG is secure.

Claim 2 *For semi-honest servers, CIP-PIR (cf. §3.3) is multi-query secure according to Defintion 1 on page 4.*

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. If all the $t-1$ servers obtain non-flip chunks, then obliviously these chunks are independent of the client request since they were generated by the servers themselves. If one of these chunks is a flip chunk sent by the client, then its value is equal to the exclusive-or of the query, the non-flip chunks chosen by the other corrupt servers, and a non-flip chunk chosen by the (non-corrupt) t-th server. That latter non-flip chunk is generated by a PRG. As in the proof of Claim 1, the security of the PRG implies that the value of this chunk is indistinguishable from a uniformly random string. Therefore so is the flip chunk received by the coalition.

Claim 3 *For malicious servers, CIP-PIR (cf. §3.3) is multi-query secure according to Defintion 1 on page 4.*

Proof: All that malicious servers can do in order to affect the information that they receive from the client is change the seeds S_i which they send to the client, for example by not choosing the seeds uniformly at random, orby resending the same seeds. Recall that each client request is translated, based on seeds received from $t-1$ servers, to a flip chunk which is sent to an additional server. The flip chunk is computed as the exclusive-or of the expansion of the $t-1$ seeds and the query. If one of the corrupt servers is the recipient of the flip chunk, then, as in the proof of Claim 2, one of the $t-1$ seeds is generated by an honest server, and therefore the expanded chunk is pseudo-random and so is the flip chunk.

If all $t-1$ non-flip chunks are chosen by corrupt servers, then the resulting flip-chunk might not hide the query (for example, if the servers 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 the flip chunk will be sent to an additional server which is not part of the coalition. (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 the flip-chunk depends on at least one legitimate seed.)[7]

B Complexity Analysis of CIP-PIR

In this section we compare the communication, computation and storage complexities of RAID-PIR [16, 17] and our new CIP-PIR scheme (cf. §3.3). We further show experimental delay times and storage overheads of CIP-PIR.

Complexities. Table 1 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. §3.4) . 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

[7] Another solution, albeit only a heuristic one, is for the client to verify that servers do not send it a seed which it previously received from any other server. This solution should work in practice but does not seem to provide provable security under the assumption of using a PRG, but rather only if we model seed expansion to be done by a random oracle.

Scheme	Communication	RTT	Server Computation (avg.)		Client Computation	Storage
RAID-PIR [17]	$n(2\sqrt{\|DB\|/8n}+\kappa/8)$	1	**Online:**	$r\|DB\|/(2n)$	$\sqrt{\|DB\|}(rn+1+1/n)$	$\|DB\|r/n$
CIP-PIR [this work]	$n(2\sqrt{\|DB\|/8n}+\kappa/8)$	2	**Offline:**	$(r-1)\|DB\|/(2n)$	$\sqrt{\|DB\|}(rn+1+1/n)$	$\|DB\|r/n+\|Q\|(\sqrt{\|DB\|}/n+\kappa)$
			Online:	$\|DB\|/(2n)$		

Table 1: 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 [16, 17] and our CIP-PIR protocol (§3.3) 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. §3.4). 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 2: Queue sizes, offline computation times, and avg. delays until the client receives the desired block of our CIP-PIR protocol (cf. §3.3)with $n=2$ servers. The offline computation is the total time for filling the empty preprocessing queue with $|Q|=10000$ entries.

Scheme	Communication		Concrete (Upload + Download)
	Upload	Download	($\|DB=16$ TByte, $n=2$, $\kappa=128$)
CIP-PIR [this work]	$n\sqrt{\|DB\|/8n}$	$n\kappa/8+n\sqrt{\|DB\|/8n}$	4096 KByte
FSS-PIR [8]	$\kappa(\log_2(\|DB\|/128)+2)$	$n\sqrt{\|DB\|/8n}$	2052 KByte
OO-PIR [34]	$2(\kappa\log_2\|DB\|+1)\log_2(\|DB\|)$	$4\sqrt{\|DB\|/8n}$	4107 KByte

Table 3: Online communication comparison of our CIP-PIR scheme with FSS-PIR [8] and Online-Offline (OO)-PIR [34] on a $|DB|$=16 TByte database with $n=2$ servers and security parameter $\kappa=128$ Bit.

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. §3.2) 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. [17] 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 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 2, 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 s the sum of the database size $|DB|$ and the queue size $|Q|$.

As already observed in Table 1, the queue size grows sublinearly with the database size, which we can also observe in Table 2. While the difference between the queue size of a 0.8 GB and 4 GB (5$\times$ larger) database is 174 MB, the difference between the 4 GB and 8 GB (only 2$\times$ 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 1), which we can approximately also see in Table 2. 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 2. 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 real-world 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. [49] 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.

Communication Comparison. Table 3 compares the *online* communication complexities of our CIP-PIR scheme with the recent FSS-PIR [8] and Online-Offline (OO)-PIR [34] 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 $|DB| = 16$ TByte 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. Its download complexity, however, is identical to our CIP-PIR scheme as each server responds with one PIR-block as well. Overall, the communication complexity of FSS-PIR improves over CIP-PIR only by factor $\approx 2\times$.

Surprisingly, the communication complexity of OO-PIR compared to CIP-PIR is almost equal although these schemes are conceptual completely different. In the $n = 2$ party setting, using the optimal blocksize for CIP-PIR, our scheme will always beat OO-PIR. If we would increase the blocksize, the improvement would be even more significant as CIP-PIR and FSS-PIR are well suited for retrieving large data or even complete files. OO-PIR, on the other hand, is very efficient for retrieving single bits or small data entries and would beat CIP-PIR easily in runtime and communication complexities. For large data items, which we need for important applications like C3 [49] and epidemiological modeling [28], CIP-PIR is more suited.

C Database Updates

The update process of CIP-PIR (§3.3) is very efficient. For this, we add dummy entries to each block in the database which serve as placeholders for new elements that are added during the update process. When a database entry is added to (replace dummy entry by new entry) or removed from (replace old entry by dummy entry) the database, the values from the (seed, value)-pair queue Q need to be updated as follows: During the update process, a subset of the database blocks change from B_i to B'_i for some indices i. In order to maintain correct (seed, value)-pairs (s,a), the servers XOR for all those indices the value $\Delta_i = B_i \oplus B'_i$ to A whenever $PRG(s)[i]$ is 1. The latter can be computed easily as we instantiate PRG with AES in CTR mode.