Sublinear Scaling for Multi-Client Private Information Retrieval

Wouter Lueks[1] and Ian Goldberg[2]

[1] Institute for Computing and Information Sciences (iCIS)
Radboud University Nijmegen
Nijmegen, The Netherlands
lueks@cs.ru.nl
[2] Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
iang@cs.uwaterloo.ca

Abstract. Private information retrieval (PIR) allows clients to retrieve records from online database servers without revealing to the servers any information about what records are being retrieved. To achieve this, the servers must typically do a computation involving the entire database for each query. Previous work by Ishai et al. has suggested using batch codes to allow a single client (or collaborating clients) to retrieve multiple records simultaneously while allowing the server computation to scale sublinearly with the number of records fetched.

In this work, we observe a useful mathematical relationship between batch codes and efficient matrix multiplication algorithms, and use this to design a PIR server algorithm that achieves sublinear scaling in the number of records fetched, even when they are requested by *distinct, non-collaborating* clients; indeed, the clients can be completely unaware that the servers are implementing our optimization. Our multi-client server algorithm is several times faster, when enough records are fetched, than existing optimized PIR severs.

As an application of our work, we show how retrieving proofs of inclusion of certificates in a Certificate Transparency log server can be made privacy friendly using multi-client PIR.

1 Introduction

Private Information Retrieval, or PIR, was introduced in the seminal work of Chor et al. in 1995 [3]. In PIR, a client wishes to retrieve information from online database servers while revealing to the database operators *no information* about what data she seeks. That this is even possible is counterintuitive, but consider the *trivial download* scheme: the database server sends the entirety of the database to the client, who searches it herself. This is clearly private, but comes at a high communication cost for large databases. Non-trivial PIR schemes aim to achieve the same level of privacy while transmitting far less data. The simplest PIR schemes assume that the database consists of an array of equal-sized blocks, and that the client knows the index of the block she wishes to retrieve. However, previous work showed that this simple query mechanism can be

used as a black box to realize more expressive database search functionality, including search by keywords [4] and private SQL queries [16].

Chor et al.'s original 1995 work showed one cannot have both information-theoretic privacy (i.e., privacy even when the server is computationally unbounded) and a sublinear (in the size of the database) communication cost if only one server is employed. However, information-theoretic PIR (IT-PIR) schemes circumvent this impossibility result by using multiple database servers and a *noncollusion* assumption—that at most a bounded number of servers (less than the total number) will collude against the client. They achieve a communication cost much smaller than the size of the database, and modern ones additionally achieve *robustness*—even if some of the servers are unresponsive, buggy, or actively malicious, the client can nonetheless retrieve her information (and identify the misbehaving servers) [2, 7, 9].

If information-theoretic privacy is not required, computational PIR (CPIR) schemes can be used. These schemes rely on computational or cryptographic assumptions to guarantee privacy against a single database server at low communication cost [14]. Devet and Goldberg [6] also recently proposed a hybrid PIR scheme that combines a CPIR scheme with an IT-PIR scheme to achieve some of the desirable properties of both, while hedging against violations of either the computational or noncollusion assumptions.

While much effort has gone into reducing the communication costs of PIR protocols, it is also important to consider the computational cost. A PIR server typically must process the entirety (or at least a significant fraction) of the database when handling each query, lest it learn information about what the client is likely *not* seeking.

Not all PIR schemes can beat the trivial download approach. Sion and Carbunar [20] found that it would always be faster to simply download the entire database than to use Kushilevitz and Ostrovsky's CPIR scheme [14]. Later, Olumofin and Goldberg [17] noted that a more modern CPIR scheme by Aguilar Melchor and Gaborit [1], as well as a number of IT-PIR schemes, are orders of magnitude faster than the trivial download scheme. However, the computation costs are still nontrivial, requiring on the order of 1 s of CPU time[3] per gigabyte of database size, for each IT-PIR query.

In order to reduce the per-query CPU cost, a number of authors have proposed *batch* techniques, in which a PIR server performs a computation over the database and a batch of simultaneous queries, resulting in less work than computing over the database once for each query separately. Henry et al. [12] propose a batching method based on *ramp schemes* particular to Goldberg's IT-PIR scheme [9], while Ishai et al. [13] use *batch codes* (discussed in more detail below) to provide multi-query computational speedups for any PIR scheme.

Both of these proposals, however, require that the *clients* construct their queries in a special way to achieve the batching speedups. This means that these approaches help only in those scenarios where single clients (or closely cooperating groups of clients) are fetching large batches of queries at the same time.

Our contributions. In this work, we address the more general case in which a PIR server wishes to process a batch of queries simultaneously, whether they were received

[3] However, this CPU time is almost completely parallelizable if multiple cores or servers are available.

all from the same client, each query from a unique client, or anything in between. We approach this problem by first observing a mathematical relationship between Ishai et al.'s method of applying batch codes to speed up IT-PIR and a special case of matrix multiplication where the left matrix has a specific structure (§3). We then generalize this observation to the case of general matrix multiplication. In doing so, we remove all restrictions on the structure of the queries to be batched. We accept a more modest batching speedup to remove the single (or coordinated) client restriction and the potentially large amount of communication induced by Ishai et al.'s method.

We apply our new technique to the setting of Certificate Transparency (§4), in which web clients fetch information about TLS certificates from log servers, but should hide from the log servers which certificate's information it fetches. This appears to be a perfect opportunity to employ PIR, but the large number of *non-cooperating* clients expected to use the system makes multi-client batching imperative. We note that while batching queries reduces the total computation time at the cost of increasing the latency for individual queries, this extra latency is not an issue in this particular application. We implement and measure our new technique on top of the open-source Percy++ PIR library [10] (§5).

While our practical improvements—a little more than a 4-fold speedup—are modest, we do offer *sublinear* scaling in the number of queries for *independent* clients, something simpler improvements cannot offer. Additionally, any other system-level optimizations can easily be used on top of our algorithmic ones.

2 Background

Our construction combines Goldberg's robust IT-PIR scheme [9] with fast matrix multiplication techniques inspired by batch codes. Therefore, we first review and compare these notions.

2.1 Goldberg's robust IT-PIR scheme

Goldberg models the database as an $r \times s$ matrix $\mathbf{D}$ over a finite field $\mathbb{F}$. Every row in $\mathbf{D}$ corresponds to a single *block* in the database; every block consists of s field elements. To request block j (non-privately) the client could simply send j to the server. However, as a first step, we express the PIR operation as a vector-matrix multiplication before producing a true privacy-friendly scheme. The client constructs the j^{th} standard basis vector e_j of $\mathbb{F}^r$ (i.e., the vector of length r with all zeros except for a 1 in the j^{th} position) and sends it to the server. The requested block j is then obtained by calculating the vector-matrix product $e_j \cdot \mathbf{D}$.

To make the query privacy friendly, the client in Goldberg's scheme creates a $(t+1)$-out-of-ℓ Shamir secret sharing [19] of this standard basis vector e_j. It sends one share to each of the ℓ database servers, which compute the vector-matrix product with the database and return the result. Lagrange interpolation of the shared vectors gives the standard basis vector; since matrix multiplication and Lagrange interpolation are linear, interpolation of the results yields the j^{th} block of the database. The secret sharing

scheme guarantees that as long as at most t servers collude, they learn nothing about the target block.

Goldberg's scheme is robust [7, 9]. It permits some of the servers to misbehave, while still enabling the client to recover her record and identify the misbehavers.

Communication cost. To read a single block, the client sends r field elements to, and receives s field elements from, each server. For a fixed database size of n field elements, it is best to select $r = s = \sqrt{n}$. Henry et al. [12] show how to build on this simple fixed-block-size PIR primitive to handle more realistic databases with variable-sized records.

Serving multiple simultaneous queries. Suppose a server receives multiple queries $v_1, \ldots, v_q$ simultaneously. It could answer them by computing the q vector-matrix products $v_i \cdot \mathbf{D}$ individually. However, it can also first group the queries into one matrix $\mathbf{Q}$ where row i consists of query v_i. Then the server computes the matrix-matrix product $\mathbf{Q} \cdot \mathbf{D}$. Row i of the result is the response to the i^{th} query.

With a naive matrix multiplication algorithm the work the server needs to do is the same in both cases: about $2qrs$ operations (qrs multiplications, and about the same number of additions). However, as we will see, using better matrix-multiplication techniques will significantly improve the situation.

Ramp scheme. Henry et al. [12] replace the Shamir secret sharing in Goldberg's PIR scheme with a *ramp scheme*. In this way, a single client can encode more information in each server request, and can retrieve q blocks instead of just 1 *without* increasing the per-server computation or communication cost at all. The large drawback to this scheme (in addition to being useful only for single clients making multiple queries, and not for multiple clients making single queries) is that it must trade some of the robustness of Goldberg's scheme for extra parallel queries, or conversely, that it requires $q - 1$ extra servers in order to maintain the same level of robustness.

2.2 Batch Codes

Batch codes can be used to answer multiple queries efficiently. The idea, proposed by Ishai et al. [13], is to encode the database in a special way, so that a single client can efficiently make multiple queries. This idea is best illustrated using an example. As in the rest of this paper, we apply the batch codes to Goldberg's IT-PIR scheme.

Suppose we want to prepare a database with r rows for two simultaneous queries. We create three separate databases: $\mathbf{D}_1$, containing the first $r/2$ rows; $\mathbf{D}_2$, containing the last $r/2$ rows; and $\mathbf{D}_3 = \mathbf{D}_1 \oplus \mathbf{D}_2$. Any two queries, say for blocks i_1 and i_2, can be answered by making at most one PIR query to each of the $\mathbf{D}_i$: if blocks i_1 and i_2 are not in the same half of the database, the queries can be answered by making one PIR query to $\mathbf{D}_1$ and one to $\mathbf{D}_2$. Suppose, on the other hand, that i_1 and i_2 are both in the first half. Then block i_1 can be retrieved directly, while block i_2 is obtained by making one query to $\mathbf{D}_2$ and one to $\mathbf{D}_3$. Taking the XOR of the latter two results yields the desired row in the first half. Two queries for the second half are handled similarly.

This procedure reduces the computational cost for the server. As we saw in the previous section, a naive method requires $4rs$ field operations; in contrast, the batch

Table 1. Summary of batch codes with parameters [13]. The subcube code is parametrized by k and $\ell \geq 2$, while the subset code is parametrized by ℓ, r' and $0 < \alpha < \frac{1}{2}$, where $w = \alpha\ell$. The parameters r and r' scale the codes to support more blocks, without essentially changing their structure.

	Subcube	Subset
Number of blocks (r)	r	$r'\binom{\ell}{w}$
Sum of subdatabase sizes (N)	$\left(\frac{\ell+1}{\ell}\right)^k r$	$r' \sum_{j=0}^{w} \binom{\ell}{j}$
Number of queries (q)	2^k	$\geq 2^w$
Number of subdatabases (m)	$(\ell+1)^k$	$\sum_{j=0}^{w} \binom{\ell}{j}$

code solution requires only $3rs$ field operations. (Again, half multiplications and half additions.)

Note that to hide which indices the client is querying she needs to make a query to each of the three parts, even if two would suffice to get the answer. This means that the client sends $\frac{3}{2}r$ elements to, and receives $3s$ elements from, each server. In the naive case she sends $2r$ elements and receives only $2s$ elements.

In general, an (r, N, q, m) batch code will take a database of r blocks and create m subdatabases, such that the total number of blocks in the subdatabases is N. The code can be used to answer q queries by making one request to each of the m subdatabases. The example we sketched before gives an $(r, \frac{3}{2}r, 2, 3)$ batch code.

Suppose we use an (r, N, q, m) batch code to speed up PIR queries to a database with r blocks, each consisting of s field elements. Let $N_1, \ldots, N_m$ be the number of blocks in the m subdatabases (so that $\sum_i N_i = N$). To make q queries a client needs to make one PIR query to each of the m subdatabases with respectively $N_1, \ldots, N_m$ blocks. The query to subdatabase i costs $2N_i s$ field operations. Therefore the total computational load on the server is $2Ns$. The client sends N group elements, and receives ms elements.

Subcube batch code Ishai et al. [13] generalize the sketch above as follows. First, instead of splitting the database into 2 parts, it can be split into ℓ parts. A final $\ell + 1^{\text{th}}$ part is added, being the XOR of all the previous parts. Again, any two items can be obtained using $\ell + 1$ queries, one to each of the subdatabases—if the two items happen to be in the same part it is necessary to retrieve and calculate the XOR of all the other items. This gives rise to an $(r, \frac{\ell+1}{\ell}r, 2, \ell + 1)$ batch code. Obviously, this is good for computation, as the server needs to do only $2\frac{\ell+1}{\ell}rs$ field operations. While the sending cost drops to $\frac{\ell+1}{\ell}r$ elements, the receiving cost rises to $(\ell + 1)s$ elements. (Note that the client always needs to retrieve the $\ell + 1$ records to protect her privacy.)

For simplicity, let us return to the case where $\ell = 2$. The scheme can be applied recursively to answer more queries. Suppose the client makes $q = 4$ queries. Group these into two pairs. Each pair can be answered by making only one query to each of the three parts $\mathbf{D}_1, \mathbf{D}_2, \mathbf{D}_3$. In total, two queries are made to each $\mathbf{D}_i$, so we can apply the above scheme again, but now on the smaller databases.

Table 2. Comparison of multi-query PIR schemes. We show counts of per-server field operations, as well as the number of field elements sent to and received from each server, the number of extra servers the scheme requires to maintain the same robustness level as for a single query, and an indication of whether independent clients can use the method, or whether all queries must be sent by a single client (or coordinated clients). The database consists of r blocks, each containing s field elements. The number of simultaneous queries, q, is assumed to be a power of 2, and much smaller than either r or s. Note that our work achieves sublinear scaling of computation in the number of queries q, while also admitting independent clients.

	Naive	Ramp [12]	Subcube Batch Codes [13]	Our work
$\mathbb{F}$ multiplications	qrs	rs	$q^{\lg((\ell+1)/\ell)}rs$	$q^{0.80735}rs$
$\mathbb{F}$ additions	$q(r-1)s$	$(r-1)s$	$\frac{(\ell^2-1)}{\ell}q^{\lg((\ell+1)/\ell)}rs$	$\frac{8}{3}q^{0.80735}rs$
Send	qr	r	$q^{\lg((\ell+1)/\ell)}r$	qr
Receive	qs	s	$q^{\lg(\ell+1)}s$	qs
Extra servers	0	$q-1$	0	0
Independent clients	✓	×	×	✓

Recursively applying this scheme gives a system that can handle $q = 2^k$ queries. Table 1 summarizes the important parameters of this scheme. By taking ℓ large, this scheme gets arbitrarily close to the optimal processing time for the server: it can answer 2^k queries with only slightly more processing than is required for a single query. However, the price is a higher communication cost for the client.

The subset batch code Ishai et al. also describe another batch code that has more favourable properties: the subset batch code. It is, however, also more complex. We only summarize the results in Table 1, and refer to their paper [13] for a full description of this scheme. The scheme is parametrized by ℓ, r', and $0 < \alpha < \frac{1}{2}$. The value w is then given by $\alpha\ell$.

It can be shown that for this code doing q queries is approximately $(1-\alpha)/(1-2\alpha)$ times more expensive than doing a single query. Thus, picking a small α brings the computational overhead for the server arbitrarily close to optimal. Contrary to the subcube codes the communication overhead is also polynomial in q, however, in practice the overhead turns out to be rather high, especially when α is small.

Consider the following example. We want α to be somewhat small, so we take $\ell = 20$ so that with $\alpha = 0.2$ we get $w = 4$. Suppose we make $q = 16$ queries, Then, $N/r = 1.279$ so the computation cost for 16 queries is only 27.9% more than for 1 query. However, we need to receive $m/q = 387$ times more data than the naive approach for $q = 16$ queries. So, using this code at low computational cost can incur extremely high communication costs.

Challenges The two main drawbacks of using batch codes for PIR are: (1) the requirement that all of the queries be generated by a single client (or by closely coop-

erating clients); and (2) the increased communication cost, which becomes especially prohibitive for large databases.

We will address both of these issues in this work. See Table 2 for a comparison of multi-query PIR schemes. Although we only list the subcube batch code and not the subset batch code in the table for conciseness, the two salient challenges listed above are the same for both types.

2.3 Matrix multiplication algorithms

Naive matrix multiplication of a matrix $\mathbf{Q}$ of size $q \times r$ with a database $\mathbf{D}$ of size $r \times s$ requires qrs multiplications and at least $q(r-1)s$ additions (although most implementations will actually use qrs additions). For two square matrices of size $n \times n$ this boils down to an $O(n^3)$ complexity.

Faster matrix multiplication algorithms exist that have an asymptotic complexity with a better exponent. In this paper we focus on Strassen's algorithm [21] because of its relative simplicity. This algorithm achieves a time complexity of $O(n^{\lg 7}) = O(n^{2.8074})$. Faster algorithms exist, such as that of Coppersmith and Winograd [5], which achieves an even better bound of $O(n^{2.3729})$. However, this comes at the cost of a much larger multiplicative constant.

Strassen's algorithm is extremely simple. It splits each matrix into four, equal-sized submatrices. A naive block-matrix multiplication of these would require 8 multiplications of the smaller sized matrices. However, using Strassen's algorithm, only 7 are needed. This technique is then applied recursively to the multiplications of the smaller matrices. (See Appendix A for more detail.)

3 Batch codes as matrix multiplication

We have seen that answering multiple PIR queries in Goldberg's protocol requires calculating the matrix-matrix product $\mathbf{Q} \cdot \mathbf{D}$, as the rows of the resulting product are exactly the responses to the given queries. At the same time, batch codes speed up this computation. Hence, batch codes are in some way implementing fast matrix multiplication. In this section we identify this relation, explain the limitations of batch codes in this application, and demonstrate the similarities with Strassen's algorithm. For simplicity of exposition (and because this is the typical case in practice), we will use $\mathbb{F}$ of characteristic 2, so that additions are just XORs.

3.1 An example

In Section 2.2 we showed how a batch code can be used to reduce two queries for the full database to three half-sized queries. In terms of matrix multiplication, the client constructs its three half-sized queries q_1, q_2, q_3 and sends them to the server. The server expresses the database $\mathbf{D}$ as a concatenation of two parts, $\mathbf{D} = \begin{pmatrix} \mathbf{D}_1 \\ \mathbf{D}_2 \end{pmatrix}$, and constructs

the matrices

$$\overline{\mathbf{Q}} = \begin{pmatrix} q_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & q_2 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & q_3 \end{pmatrix}_{3\times\frac{3}{2}r} \text{ and } \mathbf{M} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \\ \mathbf{I} & \mathbf{I} \end{pmatrix}_{\frac{3}{2}r\times r}, \text{ so that } \mathbf{M}\cdot\mathbf{D} = \begin{pmatrix} \mathbf{D}_1 \\ \mathbf{D}_2 \\ \mathbf{D}_1 \oplus \mathbf{D}_2 \end{pmatrix}_{\frac{3}{2}r\times s},$$

where $\overline{\mathbf{Q}}$ is the block-diagonal matrix of the queries, $\mathbf{I}$ is the identity matrix, and $\mathbf{M}$ is the matrix form of the batch code (representing the linear combinations of the parts that make up the resulting subdatabases). (Note that we annotate the matrices with their dimensions.) The server now computes the linear combinations of the parts, $\mathbf{M}\cdot\mathbf{D}$, and multiplies queries $\overline{\mathbf{Q}}$ by them. This results in the familiar response structure

$$\overline{\mathbf{Q}}_{3\times\frac{3}{2}r}\cdot\mathbf{M}_{\frac{3}{2}r\times r}\cdot\mathbf{D}_{r\times s} = \overline{\mathbf{Q}}_{3\times\frac{3}{2}r}\cdot\begin{pmatrix} \mathbf{D}_1 \\ \mathbf{D}_2 \\ \mathbf{D}_1 \oplus \mathbf{D}_2 \end{pmatrix}_{\frac{3}{2}r\times s} = \begin{pmatrix} q_1\cdot\mathbf{D}_1 \\ q_2\cdot\mathbf{D}_2 \\ q_3\cdot(\mathbf{D}_1\oplus\mathbf{D}_2) \end{pmatrix}_{3\times s}.$$

After receiving the results, the client combines the three rows as appropriate to recover the answers to her two original queries.

3.2 General batch codes as matrix multiplication

When using general batch codes to speed up PIR we see a similar structure. Recall that an (r, N, q, m) batch code can answer q queries to a database of r rows by splitting the computation across m subdatabases $\mathbf{K}_1, \ldots, \mathbf{K}_m$ containing a total of N rows. In the preceding example—an $(r, \frac{3}{2}r, 2, 3)$ batch code—these subdatabases were $\mathbf{K}_1 = \mathbf{D}_1$, $\mathbf{K}_2 = \mathbf{D}_2$ and $\mathbf{K}_3 = \mathbf{D}_1 \oplus \mathbf{D}_2$. For general batch codes these subdatabases will be more complicated linear combinations of the parts $\mathbf{D}_i$. The $N \times r$ matrix $\mathbf{M}$ represents these linear combinations.

Again, the client first uses the batch code to convert her q queries into m subqueries $q_1, \ldots, q_m$, where each q_i is a row vector of length equal to the number of rows in $\mathbf{K}_i$. She sends these to the server. The server constructs the linear combinations of the parts, $\mathbf{M}\cdot\mathbf{D}$, and applies the queries to them

$$\overline{\mathbf{Q}}_{m\times N}\cdot(\mathbf{M}_{N\times r}\cdot\mathbf{D}_{r\times s}) = \begin{pmatrix} q_1 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & q_m \end{pmatrix}_{m\times N}\cdot(\mathbf{M}_{N\times r}\cdot\mathbf{D}_{r\times s}).$$

The server can quickly compute this product using the block-diagonal structure of $\overline{\mathbf{Q}}$. The result is an m-row response. The client can combine those m rows to produce her desired q blocks.

Note that it is the special structure of $\mathbf{Q} = \overline{\mathbf{Q}}\cdot\mathbf{M}$ that enables the server to speed up the matrix multiplication $\mathbf{Q}\cdot\mathbf{D}$. In the PIR setting, this necessitates that $\mathbf{Q}$ be produced by a single client, or by cooperating clients.

3.3 Comparison with Strassen's algorithm

Strassen's algorithm is similar to the matrix multiplication form of the batch codes above. In particular it also

1. partitions the database into parts $\mathbf{D}_i$,
2. forms linear combinations of the $\mathbf{D}_i$ to construct the subdatabases $\mathbf{K}_i$,
3. multiplies parts of the queries with the subdatabases, and
4. computes linear combinations of the products to produce the final result.

However, there are also differences. First, batch codes require the queries to be preprocessed by the client, or alternatively that the query matrix $\mathbf{Q}$ is given by $\overline{\mathbf{Q}} \cdot \mathbf{M}$ as above. Strassen's algorithm, on the other hand, works with any matrix $\mathbf{Q}$. Second, batch codes are essentially one-dimensional; as a result, steps 1, 2 and 3 above for batch codes operate only on complete rows, while Strassen's algorithm *subdivides* and takes *linear combinations* of rows in addition to taking *subsets* of rows (in both $\mathbf{Q}$ and $\mathbf{D}$).

While Strassen's algorithm has a higher server computational cost than batch codes, the fact that Strassen's algorithm can deal with *any* matrix $\mathbf{Q}$ is of tremendous benefit. In our PIR setting, this means that clients do not need to coordinate their queries. Indeed, they do not need to be aware that the server is implementing this optimization at all.

4 Application of multi-client PIR: Certificate Transparency

We now examine an application where multi-client PIR is particularly useful: Certificate Transparency. Websites use digital certificates to tie possession of a particular private key to their domain name. These certificates are signed by certificate authorities (CAs). To verify the validity of a website the user's browser checks that a CA it trusts signed the certificate (or that there is a certificate chain from a trusted CA leading to the certificate). Events in recent years, like the hack of the Dutch CA DigiNotar [8], have shown that CAs cannot be trusted unconditionally. When a CA is compromised it can be used to issue false certificates that allow third parties to eavesdrop on the communication between a user and a website. The browser will not detect this as long as the compromised CA is still trusted. Certificate Transparency, as described in RFC 6962 [15], aims to detect wrongly issued certificates in a timely manner without introducing extra trust assumptions. It roughly works as follows.

1. Before a certificate is issued it is recorded by one or more *log servers*. Each of these log servers creates a signed certificate timestamp (SCT) for this certificate and will eventually add the certificate to an append-only data structure.
2. When presenting a certificate the website will also send along the SCTs from the log servers. The browser will verify that at least one trusted log server signed the certificate description.
3. The following consistency checks are done asynchronously:
   (a) An auditor, usually the browser, will check that the log server signed certificates do indeed appear in the append-only log of the log server.
   (b) Auditors and monitors check that the logs are consistent; i.e., that no certificates have been changed or retroactively inserted into the log.

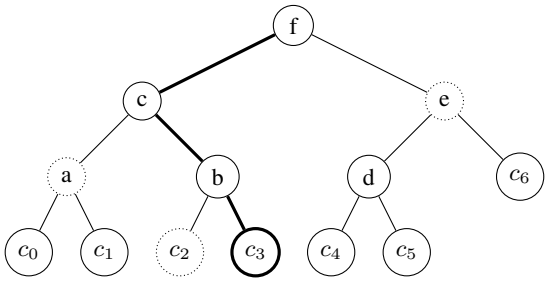

Fig. 1. An example Merkle hash tree for 7 certificates $c_0, \ldots, c_6$ encoded into the leaves. For certificate c_3 the proof of inclusion consists of all the dotted nodes: c_2, a and e. This proof can be checked as follows. First, calculate the hash of the certificate to get c_3. Then, b is the hash of c_2 and c_3; c is the hash of a and b; and, finally, f is the hash of c and e. If the calculated root f matched the signed root the auditor is convinced that the tree contains c_3.

(c) CAs and webservers monitor the log to detect inconsistencies such as two certificates, by different CAs, for the same domain.

It is essential that the first check is done, because monitors can only detect falsely issued certificates when they appear in the log. However, the first check also reveals to the log server which websites the user is visiting. We will use multi-client PIR to allow many independent clients to query a log server for the proofs of inclusion of certificates.

4.1 Proving that a certificate is included in the log

The certificates are recorded in a Merkle hash tree. A Merkle hash tree is a binary tree, in which every leaf contains the hash of a certificate, while every internal node contains the hash of its children. The root then captures information about all the children. Periodically, log servers will add all the newly logged certificates to the tree and sign the new root.

The number of leaves, n, determines the structure of the Merkle hash tree. Let k be the largest power of 2 smaller than n, so that $k < n \leq 2k$. Then the left subtree of the Merkle hash tree of n nodes is the full binary tree with k leaves, while the right subtree is the Merkle hash tree of the remaining $n - k$ nodes. See Figure 1 for an example.

This format allows log servers to construct a proof of inclusion of a certificate for an auditor. The auditor already has the certificate, and thus also the leaf corresponding to the certificate. The log server gives the auditor those node hashes needed to recalculate the root of the tree starting with the certificate. The extra nodes needed for this proof are all the siblings on the path from the leaf to the root; see Figure 1. Finally, the auditor compares the calculated root with the signed root from the server.

The length of the proof is no larger than the height of the tree. Therefore, the size of the proof grows only logarithmically in the number of leaves. The specification requires SHA256 as the hash function for the internal nodes, so a node contains 32 bytes of data.

4.2 The number of SSL certificates

To determine the feasibility of retrieving the proofs of inclusion using PIR, we need to estimate the number of active and valid SSL certificates—it does not make much sense to retrieve proofs for expired certificates. We estimate the number of SSL certificates based on the following sources.

EFF SSL Observatory The EFF SSL observatory[4] observed about 1.4 million valid certificates in 2010.

Public Netcraft Data In their public sample of May 2013,[5] Netcraft claimed that Symantec at that time had produced more than one third of all certificates. In their April, 2012 press release[6] Symantec quotes $811,511$ installed certificates. This gives an estimate of approximately 2.4 million certificates in 2012.

Pilot CT server As of early July 2014, Google's pilot certificate transparency server had logged about 4.5 million certificates.[7] It is not clear how reliable this number is, since currently there is no incentive to add all certificates to this list. Also, the log is append-only, so this number is probably higher than it should be.

Given these data points, we estimate that the number of valid SSL certificates is currently around 2^{22} or approximately 4 million.

4.3 Retrieving proofs of inclusion using PIR

To make privacy-friendly retrieval of the proofs of inclusion possible, we store the proofs as records in a database and use PIR to retrieve them. To retrieve the proof, the client needs to know in which record the proof is stored. In the original system, the proof is usually retrieved from the log server by using the hash of the certificate itself, but that would violate our privacy requirements. Instead, we propose that webserver provides the record indices to the clients (it is not possible to include these in the X.509 certificate as the index is not yet known when the certificate is created). Alternatively, an index structure such as a B+ tree could be used in the typical way that PIR lookups by keywords are done [4, 16].

To check a proof, the auditor needs three things: the certificate itself, the list of sibling hashes, and the signed root. We assume that the auditor has already retrieved the certificate in question. The signed root can be directly retrieved as it does not give any information about the specific certificate. Therefore, the proofs that are stored in the database consist solely of the hashes that help in reconstructing the signed root. We will next consider how these proofs are stored in the database.

---

[4] `https://www.eff.org/observatory`

[5] `http://www.netcraft.com/internet-data-mining/ssl-survey/`, accessed July 2014

[6] `http://investor.symantec.com/investor-relations/press-releases/press-release-details/2012/Symantec-Achieves-Highest-Number-of-SSL-Certificates-Issued-Globally/default.aspx`

[7] Obtained by querying the server's API: `https://ct.googleapis.com/pilot/ct/v1/get-sth`

Storing proofs in the PIR database We first count the number of proofs that need to be stored. The log is append-only, and will therefore keep growing. However, expired certificates can safely be removed from the database of proofs. Regular clients will not query for expired certificates, so a fallback to identifying methods is not a problem. Therefore, we assume that the database only contains proofs for valid certificates. We estimated this to be about 2^{22} proofs.

In the following we consider a tree containing $2^{\ell-1} < n \leq 2^{\ell}$ items. The length of the inclusion proofs in such a Merkle tree is at most ℓ hashes. However, it can be less; for example, the inclusion proof of c_6 in Figure 1 consists only of the nodes d and c. For simplicity, we allocate the full ℓ hashes for every proof in the database, resulting in proofs of 32ℓ bytes.

Goldberg's PIR scheme is most efficient when the number of blocks equals the number of field elements per block [12]. It thus makes sense to bundle multiple proofs into a single block (as the number of proofs is exponential in ℓ, while the length of a proof is only linear in ℓ). The location of a proof is then given by its block, and its index within the block (the size of the tree fixes the length of the proofs). When this location is provided by the webserver it should remain fixed while the certificate is valid.

Given the size of a proof and the estimated number of valid SSL certificates we get a storage requirement of $32 \cdot 22 \cdot 2^{22} \approx 0.7 \cdot 2^{32}$ bytes, or about 3 GiB. Therefore, in the following section, we evaluate our algorithm on databases of sizes 1–4 GiB.

We also note that, if the client is willing to reveal a subset of the database that contains the certificate she seeks, she can reduce the server's computational load at the cost of revealing some information about her query [18]. While trivially downloading the entire subset is one approach, PIR offers a lower communication cost—only about one *block* of information is sent to and from each PIR server—without leaking information about which certificate within the entire subset the client is querying for.

5 Implementation and Evaluation

We implemented fast matrix multiplication using Strassen's algorithm as an extension to the Percy++ open-source PIR library [10]. We implemented Strassen for the small fields $GF(2^8)$ and $GF(2^{16})$ as well as the integers modulo p. All measurements were taken in Ubuntu 12.04.4 LTS running on a machine with eight Intel(R) Xeon(R) E7-8870 CPUs, but each PIR server, which used only one core, was assigned to a different CPU.

5.1 Implementation

Strassen's algorithm works perfectly when multiplying matrices where all the dimensions are powers of two. In the PIR setting, however, this need not hold. Whenever one or more of the three matrix dimensions (q, r, or s) is odd, we split off the single excess row(s) and/or column(s) and use the naive matrix multiplication algorithm for those products. The resulting dimensions are all even, so that we can do another Strassen recursive step.

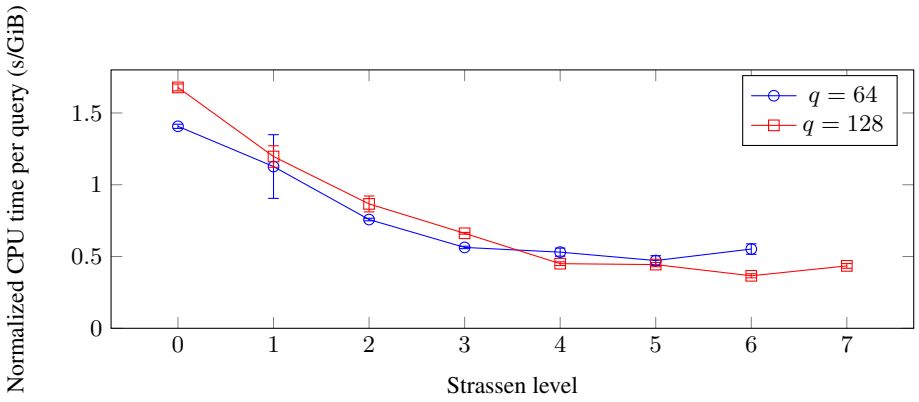

Fig. 2. Normalized CPU time per query (seconds per GiB of database size) for a 1 GiB database consisting of 32768 32768-byte records over $GF(2^8)$. We plot different Strassen levels (i.e., depth of recursion in the Strassen algorithm) and two numbers of queries, 64 and 128. Consider the $q = 64$ case. After 6 Strassen steps, the problem size has been reduced to a 1×512 matrix times a 512×512 matrix, and the algorithm bottoms out. In both cases it is better to skip this final reduction step. Error bars are shown, but most are small, and may be difficult to see.

Dealing with dimensions that are non-powers of two can be costly. For example, a dimension of $2^k - 1$ will incur extra calculations at every step, resulting in a larger computation time than if the dimension were 2^k instead. Hence, our algorithm is designed to dynamically increase the number of queries (by inserting a dummy all-zeroes query) if this yields better performance.

Every recursion step yields a small overhead. Part of this is mitigated by not allocating memory every time, but at small sizes the overhead still trumps the gain possible. We have analyzed when this happens; see Figure 2 for an example. We then tuned our implementation to stop recursing at the optimal depth.

5.2 Experiments

For q less than about 165, the number of additions in Strassen's algorithm is slightly larger than for the naive algorithm due to the multiplicative constant of $\frac{8}{3}$ (see Table 2). However, as explained above, every recursive Strassen step reduces the number of multiplications by a factor of $7/8$ or 12.5%, starting with the very first. The small fields and the integers modulo p have in common that multiplication is a lot more expensive than addition; therefore, we expect that even one or two recursive steps of Strassen's algorithm would have a measurable effect on the performance, and the measurements in Figure 2 bear this out. The initial dimensions of the problem dictate how many recursive steps of Strassen's algorithm can be applied, as each dimension is cut in half at each step. In practice, we expect that it is the number of queries that is the limiting factor (that is, q will be much smaller than either r or s), so that is what we focus on.

Figure 3 compares the performance of our new algorithm with the one in the 0.9.0 release of Percy++. All measurements are done over $GF(2^8)$, as that is the most ef-

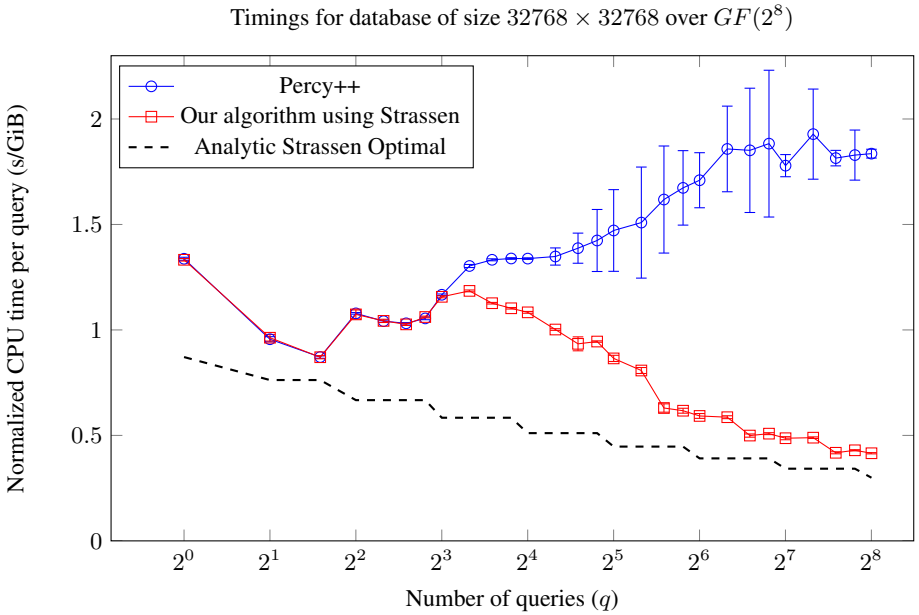

Fig. 3. Comparison between the original $GF(2^8)$ PIR implementation in Percy++ and our new version using Strassen. For the first part of the graph, the new and original algorithms give the same results, as the Strassen code is not invoked for small problem sizes. We can also easily see the effect of the hand-optimized loop in Percy++ for handling $q \leq 3$. We also compare our algorithm to the best theoretical improvements that using Strassen's algorithm can provide, using the fastest per-query time of the original Percy++ code ($q = 3$) as a reference point. Error bars are shown, but most are small, and may be difficult to see. The peak memory usage of our algorithm (for $q = 256$) was 1422 MiB, whereas Percy++ used 1060 MiB.

ficient field supported by Percy++. We notice that Percy++ slows down considerably when more queries are used (we suspect cache issues may be to blame for this, but it was a completely repeatable effect). Our scheme does not suffer from this problem, and indeed produces the desired *decrease* in per-query cost as the number of queries increases. We observe a 4.4-fold performance improvement over Percy++ for $q = 256$ simultaneous queries.

We have also drawn the analytical improvements we expect from using Strassen's algorithm. Figure 3 shows the theoretical bound of the optimal Strassen gain that would be possible, measured against the *fastest* per-query time (obtained at $q = 3$) measured for the original Percy++ code. For example, for 256 queries, this gain would be $(\frac{7}{8})^8 \approx 0.344$. We see that we are quite close to this value, even though we always skip the final Strassen step.

For each Strassen recursion step, our algorithm incurs an extra subproblem of 1/4 the size, which needs to be stored in memory. The extra memory consumption as a result of this is at most a factor of $\frac{1}{4} + (\frac{1}{4})^2 + \cdots = \frac{1}{3}$. This is confirmed by the memory usage given in Figures 3 and 4.

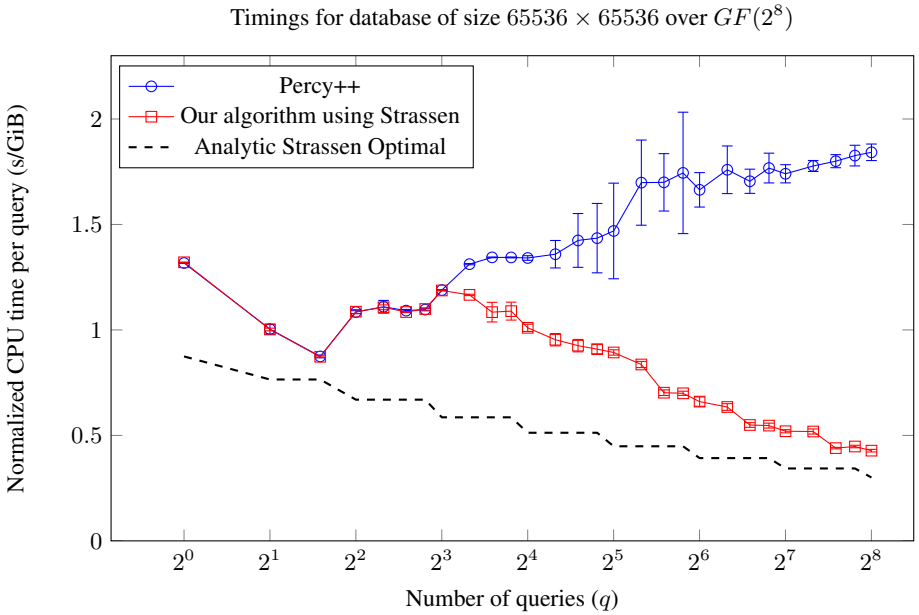

Fig. 4. We repeat the experiment of Figure 3, but with a 4 GiB database consisting of 65536 65536-byte records. The peak memory usage of our algorithm (for $q = 256$) was 5556 MiB, whereas Percy++ used 4148 MiB.

Certificate Transparency. Figure 4 shows performance measures for a 4 GiB database, a size that nicely matches up with a log server's database of inclusions proofs. Again we see that using Strassen's algorithm gives a significant performance gain over just using single queries.

While batching queries results in a significantly lower processing time per query, the latency does increase. However, this is not a problem for auditing proofs of inclusion, as they are performed asynchronously. In particular, the goal is to detect misbehaving log servers, and not to protect users against falsely issued certificates directly. Some latency is therefore acceptable.

If a lower latency is required, the algorithm can easily be parallelized. Each of the seven Strassen subproblems is completely independent from the others, and creating these and recombining the result is very cheap. While we did not implement parallelization, we expect the overhead of doing so to be extremely small.

6 Conclusions

In this paper we showed how we can significantly speed up PIR queries if we allow the server to batch queries, and answer them simultaneously. Such an idea was proposed earlier in the setting of batch codes, but that proposal required coordination among the clients, which is not desirable in the PIR setting.

We analyzed batch codes in the setting of Goldberg's PIR scheme and have shown that essentially they provide a fast method for doing matrix multiplication under specific constraints on the matrices. However, since multi-client PIR in Goldberg's scheme is essentially a matrix multiplication, we can use our method to obtain sublinear scaling in the number of queries *without* requiring the queries to have been created by a single client or cooperating clients.

We described how multi-client PIR can be used to make certificate transparency more privacy friendly. We implemented Strassen's algorithm as part of Percy++ and have shown that we indeed manage to get a significant speedup when batching multi-client queries. While further system-level optimizations to Percy++ (which is already heavily optimized) might conceivably give comparable speedups in absolute terms, these will almost surely be a constant factor, whereas our algorithmic improvements *increase* with the number of simultaneous queries. Furthermore, any such system-level optimizations are likely to be able to be combined with our algorithmic improvements to yield compounded benefits.

Our implementation is open source and has been incorporated into the 1.0 release of Percy++ [11].

Acknowledgements. We thank the anonymous reviewers and Ben Laurie for their helpful feedback. This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the research program Sentinels as project 'Revocable Privacy' (10532). Wouter Lueks is a member of the Privacy and Identity Lab (PI.lab). Sentinels is being financed by Technology Foundation STW, the Netherlands Organization for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs. The PI.lab is funded by SIDN.nl (`http://www.sidn.nl`). This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo.

References

1. Aguilar Melchor, C., Gaborit, P.: A Lattice-Based Computationally-Efficient Private Information Retrieval Protocol. In: Western European Workshop on Research in Cryptology (2007)
2. Beimel, A., Stahl, Y.: Robust Information-Theoretic Private Information Retrieval. Journal of Cryptology 20(3), 295–321 (2007)
3. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private Information Retrieval. In: 36th Annual IEEE Symposium on Foundations of Computer Science. pp. 41–50 (1995)
4. Chor, B., Gilboa, N., Naor, M.: Private Information Retrieval by Keywords. Technical Report TR CS0917, Department of Computer Science, Technion, Israel (1997)
5. Coppersmith, D., Winograd, S.: Matrix Multiplication via Arithmetic Progressions. Journal of Symbolic Computation 9(3), 251–280 (1990)
6. Devet, C., Goldberg, I.: The Best of Both Worlds: Combining Information-Theoretic and Computational PIR for Communication Efficiency. In: 14th Privacy Enhancing Technologies Symposium. pp. 63–82 (2014)
7. Devet, C., Goldberg, I., Heninger, N.: Optimally Robust Private Information Retrieval. In: 21st USENIX Security Symposium (2012)

8. Fox-IT BV: Black Tulip: Report of the investigation into the DigiNotar Certificate Authority breach (Aug 2012)
9. Goldberg, I.: Improving the Robustness of Private Information Retrieval. In: 28th IEEE Symposium on Security and Privacy. pp. 131–148 (2007)
10. Goldberg, I., Devet, C., Hendry, P., Henry, R.: Percy++ project on SourceForge. `http://percy.sourceforge.net` (2013), version 0.9.0. Accessed Sept. 2014
11. Goldberg, I., Devet, C., Lueks, W., Yang, A., Hendry, P., Henry, R.: Percy++ project on SourceForge. `http://percy.sourceforge.net/` (2014), version 1.0. Accessed Nov. 2014
12. Henry, R., Huang, Y., Goldberg, I.: One (Block) Size Fits All: PIR and SPIR with Variable-Length Records via Multi-Block Queries. In: 20th Annual Network and Distributed System Security Symposium (2013)
13. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Batch Codes and Their Applications. In: 36th ACM Symposium on Theory of Computing. pp. 262–271 (2004)
14. Kushilevitz, E., Ostrovsky, R.: Replication Is Not Needed: Single Database, Computationally-Private Information Retrieval. In: 38th Annual IEEE Symposium on Foundations of Computer Science. pp. 364–373 (1997)
15. Laurie, B., Langley, A., Kasper, E.: Certificate Transparency. RFC 6962 (Experimental) (Jun 2013), `http://www.ietf.org/rfc/rfc6962.txt`
16. Olumofin, F., Goldberg, I.: Privacy-preserving Queries over Relational Databases. In: 10th International Privacy Enhancing Technologies Symposium. pp. 75–92 (2010)
17. Olumofin, F., Goldberg, I.: Revisiting the Computational Practicality of Private Information Retrieval. In: 15th International Conference on Financial Cryptography and Data Security. pp. 158–172 (2011)
18. Olumofin, F., Tysowski, P.K., Goldberg, I., Hengartner, U.: Achieving Efficient Query Privacy for Location Based Services. In: 10th Privacy Enhancing Technologies Symposium. pp. 93–110 (2010)
19. Shamir, A.: How to share a secret. Communications of the ACM 22(11), 612–613 (1979)
20. Sion, R., Carbunar, B.: On the Computational Practicality of Private Information Retrieval. In: 14th Network and Distributed Systems Security Symposium (2007)
21. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik 13(4), 354–356 (1969)

A Strassen's algorithm

Strassen's algorithm is best explained by looking at matrix multiplication from a block-matrix perspective. For simplicity, assume that all matrices have size $n \times n$ where n is even. If

$$\mathbf{Q} = \begin{pmatrix} \mathbf{Q}_{11} & \mathbf{Q}_{12} \\ \mathbf{Q}_{21} & \mathbf{Q}_{22} \end{pmatrix} \quad \text{and} \quad \mathbf{D} = \begin{pmatrix} \mathbf{D}_{11} & \mathbf{D}_{12} \\ \mathbf{D}_{21} & \mathbf{D}_{22} \end{pmatrix},$$

then the matrix product $\mathbf{R} = \mathbf{Q} \cdot \mathbf{D}$ is given by

$$\mathbf{R} = \begin{pmatrix} \mathbf{R}_{11} & \mathbf{R}_{12} \\ \mathbf{R}_{21} & \mathbf{R}_{22} \end{pmatrix},$$

where

$$\begin{aligned}
\mathbf{R}_{11} &= \mathbf{Q}_{11} \cdot \mathbf{D}_{11} + \mathbf{Q}_{12} \cdot \mathbf{D}_{21} \\
\mathbf{R}_{12} &= \mathbf{Q}_{11} \cdot \mathbf{D}_{12} + \mathbf{Q}_{12} \cdot \mathbf{D}_{22} \\
\mathbf{R}_{21} &= \mathbf{Q}_{21} \cdot \mathbf{D}_{11} + \mathbf{Q}_{22} \cdot \mathbf{D}_{21} \\
\mathbf{R}_{22} &= \mathbf{Q}_{21} \cdot \mathbf{D}_{12} + \mathbf{Q}_{22} \cdot \mathbf{D}_{22}.
\end{aligned}$$

It thus reduces to 8 matrix multiplications of size $n/2$. In Strassen's algorithm the following 7 matrix products are calculated first (note that in fields of characteristic 2, the $+$ and $-$ operations are of course the same):

$$\begin{aligned}
\mathbf{M}_1 &= (\mathbf{Q}_{11} + \mathbf{Q}_{22}) \cdot (\mathbf{D}_{11} + \mathbf{D}_{22}) \\
\mathbf{M}_2 &= (\mathbf{Q}_{21} + \mathbf{Q}_{22}) \cdot \mathbf{D}_{11} \\
\mathbf{M}_3 &= \mathbf{Q}_{11} \cdot (\mathbf{D}_{12} - \mathbf{D}_{22}) \\
\mathbf{M}_4 &= \mathbf{Q}_{22} \cdot (\mathbf{D}_{21} - \mathbf{D}_{11}) \\
\mathbf{M}_5 &= (\mathbf{Q}_{11} + \mathbf{Q}_{12}) \cdot \mathbf{D}_{22} \\
\mathbf{M}_6 &= (\mathbf{Q}_{21} - \mathbf{Q}_{11}) \cdot (\mathbf{D}_{11} + \mathbf{D}_{12}) \\
\mathbf{M}_7 &= (\mathbf{Q}_{12} - \mathbf{Q}_{22}) \cdot (\mathbf{D}_{21} + \mathbf{D}_{22}).
\end{aligned}$$

The matrix product is then given by:

$$\begin{aligned}
\mathbf{R}_{11} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{R}_{12} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{R}_{21} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{R}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6.
\end{aligned}$$

Using this algorithm, only 7 matrix multiplications of size $n/2$ are necessary. Applying this trick recursively gives a complexity of $O(n^{\lg 7})$.