Optimally Robust Private Information Retrieval*

Casey Devet *Ian Goldberg*
University of Waterloo
{cjdevet,iang}@cs.uwaterloo.ca

Nadia Heninger
University of California, San Diego
nadiah@cs.ucsd.edu

Abstract

We give a protocol for multi-server information-theoretic private information retrieval which achieves the theoretical limit for Byzantine robustness. That is, the protocol can allow a client to successfully complete queries and identify server misbehavior in the presence of the maximum possible number of malicious servers. We have implemented our scheme and it is extremely fast in practice: up to thousands of times faster than previous work. We achieve these improvements by using decoding algorithms for error-correcting codes that take advantage of the practical scenario where the client is interested in multiple blocks of the database.

1 Introduction and related work

Private information retrieval (PIR) is a way for a client to look up information in an online database without letting the database servers learn the query terms or responses. A simple if inefficient way to do this is for the database server to send a copy of the entire database to the client, and let the client look up the information for herself. This is called *trivial download*. The goal of PIR is to transmit less data while still protecting the privacy of the query. PIR is a fundamental building block for many proposed privacy-sensitive applications in the literature, including patent databases [2], domain name registration [28], anonymous email [33], and improving the scalability of anonymous communication networks [26].

The simplest kind of query one can make with PIR is to consider the database to be composed of a number of blocks of equal size, and to retrieve a particular block from the database by its absolute position [10]. Although this simple type of query does not appear to be very useful in practice, it turns out that it can be used as a black-box building block to construct more complex and useful queries, such as searching for keywords [9] or private SQL queries [28].

PIR protocols can be grouped into two classes corresponding to the security guarantees they provide. One class is *computational* PIR [8], in which the database servers can learn the client's query if they can apply sufficient computational power to break a particular cryptographic system. The other class of protocols — those we will consider in this work — is *information-theoretic* PIR [10, 11], in which no amount of computation will allow the reconstruction of the client's query. In these protocols, the query is protected by splitting it among multiple database servers. (Chor et al. [10] show that information-theoretic PIR with less data transfer than the trivial download scheme is impossible with only one server.) As is common in many distributed privacy-enhancing technologies, such as mix networks [7], Tor [14], or some forms of electronic voting [6], we must assume that some fraction of the servers above some threshold are not colluding against the client.

While much of the theoretical work on PIR focuses strictly on minimizing the amount of data transferred [15, 38], in a practical setting we must take other aspects, particularly the computational performance, into account. In 2007, Sion and Carbunar [36] opined that, given trends in computational power and network speeds, it would always be faster to send the whole database to the client than to use PIR to process it. However, they only considered one kind of computational PIR [23] in their analysis.

In fact, recent work by Olumofin and Goldberg [29] demonstrates that a more recent computational PIR scheme by Aguilar Melchor and Gaborit [1] is an order of magnitude faster than trivial download, while information-theoretic (IT) PIR can be two to three orders of magnitude faster. These PIR protocols are well matched to deployment on mobile clients as they require low data transfer, low client-side computation, and moderate server-side computation [30]. For example, to retrieve one 32 KiB block from a 1 GiB database, an IT-

*This is an extended version of our conference paper, which appeared at the 2012 USENIX Security Symposium. [13]

PIR client would send one block of data to, and receive one block of data from, each server. The servers each perform about 1.4 CPU seconds of computation, and the client performs about 140 ms of computation.

1.1 Byzantine robustness

An important practical consideration with multi-server PIR is how to deal with servers that *do not respond* to a client's queries, or that *respond incorrectly*, either through malice or error. These are respectively termed the *robustness* and *Byzantine robustness* problems.

The main result of this paper is to improve the Byzantine robustness of information-theoretic PIR. In order to guarantee information-theoretic PIR, one must have multiple servers in the protocol; Byzantine robustness guarantees that the protocol still functions correctly even if some of the servers fail to respond or give incorrect or malicious responses. Byzantine robustness makes no assumptions on the type of errors that can appear—the model covers spurious or random errors as well as malicious interference—and the bounds are given in terms of the number of servers which ever give incorrect responses. The client must still be able to determine the answer to her query, even when some number of the servers fail to respond, or give incorrect answers; further, in the latter case, the client would like to learn *which* servers misbehaved so that they can be avoided in the future. (In the single-server case, the owner of the database can provide a cryptographic signature on each block in order to ensure integrity, as PIR-Tor [26] does. Without computational assumptions or some kind of shared secret, it does not make much sense to consider robustness or Byzantine robustness in a single-server PIR setting.)

Beimel and Stahl [3, 4] were the first to consider robustness and Byzantine robustness for PIR. Consider an ℓ-server information-theoretic PIR setting, where only k of the servers respond, v of the servers respond incorrectly, and the system can withstand up to t colluding servers without revealing the client's query (t is called the *privacy level*). (This is termed "t-private v-Byzantine robust k-out-of-ℓ PIR".) Then the protocol of Beimel and Stahl works when $v \leq t < k/3$. Under those conditions, the protocol will always output to the client a unique block, which will be the correct one; this is called *unique decoding*.

In 2007, Goldberg [19] observed that by allowing for the possibility of *list decoding* — that is, that the protocol may sometimes output a small number of blocks instead of just one — the privacy level and the number of misbehaving servers can be substantially increased, up to $t < k$ and $v < k - \lfloor\sqrt{kt}\rfloor$. He also showed that in many scenarios, the probability of more than one block being output by the protocol is vanishingly small, while in others, one can employ standard techniques to convert list decoding to unique decoding [25] at the cost of slightly increasing the size of the database. The communication overhead of Goldberg's protocol is $k+\ell$; that is, to retrieve one block of data (say b bits), the protocol transfers a total of $(k+\ell)\,b$ bits, for the optimal choice of block size b.

1.2 Our contributions

- We change only the client side of Goldberg's 2007 protocol to improve its Byzantine robustness from $v < k - \lfloor\sqrt{kt}\rfloor$ to $v < k - t - 1$, which is the theoretically maximum possible value. Depending on the deployment scenario, the communication overhead of our protocol ranges from a factor of $k+\ell$ to a maximum of $v(k+\ell)$.

- Our protocol is considerably faster than Goldberg's protocol for many reasonable parameter choices. We implemented our protocol on top of Goldberg's open-source Percy++ [18] distribution and find that our new protocol can be up to *3–4 orders of magnitude* (thousands of times) faster than the original in reconstructing the correct response to a query in the presence of Byzantine servers.

The robustness and efficiency improvements to the PIR protocol given in this paper mean that recovering from Byzantine errors even in an extremely adversarial or noisy setting is not just academically feasible, but is completely reasonable for user-facing applications.

Goldberg's protocol uses Shamir secret sharing to hide the query; since Shamir secret sharing is based off of polynomial interpolation, the problem of recovering the response in the case of Byzantine failures corresponds to noisy polynomial reconstruction, which is exactly the problem of decoding Reed-Solomon codes. The theoretical contribution of this work is to observe that the practical setting of clients performing multiple queries allows us to use sophisticated decoding algorithms that can decode multiple queries *simultaneously* and achieve an enormous improvement in both performance and the level of robustness.

1.3 Organization

The remainder of the paper is organized as follows. In Section 2 we will introduce the tools that we need to present our protocol: Shamir secret sharing, Reed-Solomon codes, and decoding algorithms for multipolynomial extensions of these codes. In Section 3 we review the PIR protocols that form the foundation for our work. We present our protocol and algorithms in Section 4, and give experimental results in Section 5. We conclude the paper in Section 6.

2 Preliminaries

2.1 Notation

We will use the following variables throughout the paper:

- ℓ denotes the total number of servers
- t is the privacy level: no coalition of t or fewer servers can learn the client's query
- k is the number of servers that respond
- v is the number of Byzantine servers that respond and h is the number of honest servers that respond (so $h+v=k$). Byzantine servers may respond with any maliciously chosen value.
- D is the database
- r is the number of blocks in the database
- s is the number of words in each database block
- w is the number of bits per word

We denote by $\mathbf{e}_j$ the standard basis vector $\langle 0,\ldots,0,1,0,\ldots,0\rangle$ where the 1 is in the jth place. $x \in_R X$ means selecting the element x uniformly at random from the space X.

2.2 Shamir secret sharing

The classic Shamir secret sharing scheme [34] allows a *dealer* to choose a secret value σ, and distribute *shares* of that secret to ℓ *players*. If t or fewer of the players come together, they learn no information about σ, but if more than t pool their shares, they can easily recover the secret. (t and ℓ are parameters of the scheme, with $t<\ell$.)

The scheme works as follows: let σ be an arbitrary element of some finite field $\mathbb{F}$ (not necessary uniformly distributed). The dealer selects ℓ arbitrary distinct non-zero indices $\alpha_1,\ldots,\alpha_\ell \in \mathbb{F}$, and selects t elements $a_1,\ldots,a_t \in_R \mathbb{F}$ uniformly at random. The dealer constructs the polynomial $f(x)=\sigma+a_1x+a_2x^2+\cdots+a_tx^t$, and gives to player i the share $(\alpha_i, f(\alpha_i)) \in \mathbb{F}\times\mathbb{F}$ for $1\le i\le \ell$. Note that the secret σ is just $f(0)$. Now any $t+1$ or more players can use Lagrange interpolation to reconstruct the polynomial f, and evaluate $f(0)$ to yield σ. However, t or fewer players learn absolutely no information about σ.

Complications arise during reconstruction, however, when some of the shares being brought together to reconstruct f are incorrect. Dealing with this case involves working with error-correcting codes, and will be discussed in Section 2.3, next.

Sharing a vector of elements in $\mathbb{F}^r$ rather than a single field element is done in the straightforward way: each coordinate of the vector is secret shared separately, using r independent random polynomials.

2.3 Error-correcting codes

We will use error-correcting codes to handle Byzantine robustness. In the case of servers that merely fail to respond, we could try to use an erasure code — an error-correcting code which can be decoded when some symbols are erased by the channel. In order to handle Byzantine failures, we will use error-correcting codes that can handle both corrupted and missing symbols. Our scheme will transform malicious errors into random errors, which will allow us to achieve much higher robustness (with high probability) than was efficiently possible before. In addition, the use of these error-correcting codes allows us to identify servers that cheat during the protocol, and not use them in the future.

The error-correcting codes that we will use in our protocol are based off of Reed-Solomon codes. [32] This is a natural choice to use with Shamir secret sharing, as they both use polynomial interpolation. If a message of length $t+1$ consists of elements $\{a_0,a_1,\ldots,a_t\}$ in some field $\mathbb{F}$ then we can define the degree-t polynomial $f(x)=a_0+a_1x+\cdots+a_tx^t$. Fix k distinct field elements $\alpha_1,\ldots,\alpha_k$. A Reed-Solomon codeword consists of the evaluations of f at each point: $\{f(\alpha_1),\ldots,f(\alpha_k)\}$.

The Berlekamp-Welch [5] algorithm can efficiently decode a Reed-Solomon codeword with up to $v<(k-t)/2$ errors, which is the theoretical maximum for unique decoding. However, if one is willing to accept the possibility of decoding to multiple valid codewords, the Guruswami-Sudan algorithm [22] improves the *decoding radius* to $v<k-\sqrt{kt}$. This is known as *list decoding*: the algorithm returns a list of all valid codewords.

2.3.1 Multi-polynomial reconstruction

The above decoding algorithms all consider the case of noisy interpolation of a single polynomial. More recently, Parvaresh and Vardy [31], and Guruswami and Rudra [21] designed codes that could be efficiently list decoded, approaching the asymptotic limit of $v<k-t-1$. These codes are based around the idea of extending the Reed-Solomon code to evaluate *multiple polynomials* simultaneously, and using clever constructions of the polynomials in order to efficiently decode the codewords. One of the main contributions of this paper is to adapt these ideas to a cryptographic setting. We cannot directly use their constructions, as their polynomials have a special structure that would make them unsuitable for secret sharing. However, using a randomized construction we can nonetheless efficiently decode such multi-polynomial codes in practice with high probability, yielding a secret sharing system robust to many errors.

The codes that we will use will reconstruct several polynomials simultaneously from noisy evaluation

points. Define m polynomials

$$f_1(x) = a_{10} + a_{11}x + \cdots + a_{1t}x^t,$$
$$\vdots$$
$$f_m(x) = a_{m0} + a_{m1}x + \cdots + a_{mt}x^t.$$

Then a codeword will consist of the evaluations of each of these polynomials at points $\alpha_1, \ldots, \alpha_k$:

$$f_1(\alpha_1), \ldots, f_m(\alpha_1),$$
$$\vdots$$
$$f_1(\alpha_k), \ldots, f_m(\alpha_k)$$

This general case is considered by Cohn and Heninger [12], who give an algorithm that heuristically reconstructs every polynomial as long as there are no more than $v < k - t^{m/(m+1)}k^{1/(m+1)}$ values of i for which the received value of some $f_p(\alpha_i)$ is incorrect. In our application to PIR, each polynomial will correspond to a column of the database matrix D, and each value α_i will correspond to a PIR server; therefore, we will be able to tolerate v dishonest servers.

2.3.2 Linear multi-polynomial decoding

The list-decoding algorithms of Guruswami-Sudan, Parvaresh-Vardy, and Cohn-Heninger all work by constructing a polynomial which vanishes to high multiplicity at the codeword. If one simply uses multiplicity one, one can obtain a "linear" variant of the Cohn-Heninger algorithm [12] which is extremely fast in practice. It reconstructs each polynomial uniquely when no more than $v \leq \frac{m}{m+1}(k-t-1)$ values of i have incorrect received values of some $f_p(\alpha_i)$. This algorithm works with high probability in practice as long as the errors are randomized. We will show later how to set up our protocol to enforce that even malicious servers can only insert random errors.

Since this linear variant is not explicitly described in their work, we provide a brief outline in Algorithm 1.

Algorithm 1 Fast multi-polynomial reconstruction

Input: km points (α_i, y_{ip}) $1 \leq i \leq k$, $1 \leq p \leq m$, degree bound t, and minimum number of correct points $h = k - v$.

Output: m polynomials $f_1, \ldots, f_m$ of degree at most t such that for at least h values of i, $f_p(\alpha_i) = y_{ip}$ for all $1 \leq p \leq m$

1: Use Lagrange interpolation to construct m polynomials f_p^* of degree at most $k-1$ s.t. $f_p^*(\alpha_i) = y_{ip}$ for each $1 \leq i \leq k$.

2: Construct the degree-k polynomial

$$N(x) = \prod_{i=1}^{k}(x - \alpha_i)$$

3: Construct the $(m+1) \times (m+1)$ polynomial matrix

$$M = \begin{bmatrix} x^t & & & & -f_1^*(x) \\ & x^t & & & -f_2^*(x) \\ & & \ddots & & \\ & & & x^t & -f_m^*(x) \\ & & & & N(x) \end{bmatrix}$$

4: Run a polynomial lattice basis row reduction algorithm on M.

5: Discard the largest-degree row in the reduced matrix. If any remaining row has degree larger than h, abort.

6: Write the remaining $m \times (m+1)$ matrix as $[\mathbf{A}|\mathbf{b}]$, where $\mathbf{A}$ is an $m \times m$ matrix, and $\mathbf{b}$ is an $m \times 1$ column vector.

7: Solve the linear system of equations

$$\left(\frac{1}{x^t}\mathbf{A}\right)\begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix} = \mathbf{b}$$

8: **return** $(f_1, \ldots, f_m)$

Polynomial lattice basis reduction. Step 4 in the algorithm uses a "polynomial lattice basis row reduction algorithm". This is an algorithm which takes as input a matrix M of *polynomials* and applies elementary row operations over the ring of polynomials to produce a matrix M' whose coefficient polynomials have minimal degree. [37] There are several polynomial-time polynomial lattice basis reduction algorithms. (This is a refreshing contrast to the case of *integer* lattices where finding exact shortest vectors is NP-hard and efficient algorithms such as LLL [24] can only obtain an exponential approximation.) The algorithm of Giorgi et al. [17] runs in time $O(\delta n^{\omega+o(1)})$ where δ is the maximum degree of the input basis, n is the dimension, and ω is the exponent of matrix multiplication. Our implementation uses the algorithm of Mulders and Storjohann [27] which runs in time $O(n^3\delta^2)$ but is much simpler and easier to implement, and yields excellent running times for the input sizes we care about.

If step 5 does not abort, then there is guaranteed to be a unique set of polynomials satisfying the requirements; that is, we are in the unique decoding case. As we will

see later, if the errors are random, this step aborts only with very low probability.

Without any imposed structure on the codeword polynomials, the Cohn-Heninger algorithm is *heuristic*; that is, they conjecture that it will succeed for sufficiently random input. The linear version that we use here is also heuristic: there are adversarial inputs on which it may fail. However, we observe in experiments (see Section 5.3) that the heuristic assumption holds with high probability for *random* inputs, which is the situation we need for our cryptographic purposes here. We conjecture based on the experimental evidence presented in Section 5.3 that the probability of failure depends only on the size of the underlying field $\mathbb{F}$. In particular, the algorithm will work with high probability for random polynomials if the errors are uncorrelated. We will see later that we can enforce this restriction in our protocol even in the case of Byzantine servers.

2.3.3 Optimality

Relating this to our PIR application, we will be able to use this algorithm to correctly decode the results of t-private PIR queries. If k servers respond to us, of which v are Byzantine (so $h = k - v$ are honest), then this algorithm will succeed with high probability after we query for m blocks, satisfying $v \leq \frac{m}{m+1}(k-t-1)$, or equivalently, $m \geq \frac{v}{h-t-1}$. That is, for m large enough, we can handle any number of Byzantine servers $v < k-t-1$. We note that this bound on v is *optimal*—if $v = k-t-1$, or equivalently, $h = t+1$, then *any* subset of $t+1$ servers' responses will form a polynomial of degree at most t. This means that the number of possible valid blocks will always be exponential, and no polynomial-time algorithm could hope to address this case.

2.4 Dynamic programming

In practice, each of the algorithms we have described above has different performance characteristics for different inputs. Thus in our implementation, we achieve the best performance by assembling all of them together into a *portfolio algorithm*. This algorithm optimistically attempts to decode a given input using Lagrange interpolation, and if that fails, uses a dynamic program with timing measurements to fall back to an optimal sequence of decoding algorithms. See Section 5.2 for more details.

3 Protocols for PIR

In this section, we will introduce the ideas from previous PIR protocols that will form the basis for our protocol.

3.1 Database queries as linear algebra

We begin with a general mathematical setting of the PIR schemes we will be considering.

Our database D is structured as an $r \times s$ matrix with r rows. Each row represents one block of the database, and consists of s words of w bits each. The database D resides on a remote server. The client wishes to retrieve one block (row) of the database from the server.

$$D = \begin{bmatrix} \mathbf{w}_{11} & \mathbf{w}_{12} & \dots & \mathbf{w}_{1s} \\ \mathbf{w}_{21} & \mathbf{w}_{22} & \dots & \mathbf{w}_{2s} \\ \vdots & \vdots & & \vdots \\ \mathbf{w}_{r1} & \mathbf{w}_{r2} & \dots & \mathbf{w}_{rs} \end{bmatrix}$$

One non-private protocol for the client to retrieve row β of the database would be to transmit the vector $\mathbf{e}_\beta$ consisting of all zeros except for a single 1 in coordinate β to the server. The server considers $\mathbf{e}_\beta$ as a row vector and computes the product $\mathbf{e}_\beta \cdot D$, which it sends back to the client.

$$\begin{bmatrix} 0 & 0 & \dots & 1 & \dots & 0 \end{bmatrix} \begin{bmatrix} \mathbf{w}_{11} & w_{12} & \dots & \mathbf{w}_{1s} \\ \mathbf{w}_{21} & \mathbf{w}_{22} & \dots & \mathbf{w}_{2s} \\ \vdots & \vdots & & \vdots \\ \mathbf{w}_{r1} & \mathbf{w}_{r2} & \dots & \mathbf{w}_{rs} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_{\beta 1} & \mathbf{w}_{\beta 2} & \dots & \mathbf{w}_{\beta s} \end{bmatrix}$$

We will show how to construct two information-theoretic PIR schemes that modify this basic scheme to retrieve blocks from the database without revealing the query or result to an adversary.

3.2 A simple PIR scheme due to Chor et al.

We next present a simple PIR scheme due to Chor et al. [10] We begin with the same setup as above. In this protocol, the words will be single bits, so $w = 1$, and D is an $r \times s$ matrix of bits. Since we will be constructing information-theoretic PIR, we will be querying more than one server. We will require that not all of the servers are colluding to reveal the client's query. Each of the $\ell \geq 2$ servers gets a copy of D.

A client wishing to retrieve block β of the database generates the basis vector $\mathbf{e}_\beta$ as above to select coordinate β. Then in order to hide this query vector from the servers, the client picks $\ell - 1$ vectors $\mathbf{v}_1, \dots, \mathbf{v}_{\ell-1}$ uniformly at random from $GF(2)^r$ (that is, $\ell - 1$ uniformly random r-bit binary strings), and computes $\mathbf{v}_\ell = \mathbf{e}_\beta \oplus (\mathbf{v}_1 \oplus \cdots \oplus \mathbf{v}_{\ell-1})$. $\mathbf{v}_\ell$ will be a uniformly random (though not independent) r-bit string, as $\ell \geq 2$.

The client sends $\mathbf{v}_i$ to server i for each $1 \leq i \leq \ell$. Server i computes the product $\mathbf{r}_i = \mathbf{v}_i \cdot D$, which is the same as

setting $\mathbf{r}_i$ to be the XOR of those blocks j in the database for which the jth bit of $\mathbf{v}_i$ is 1. Each server i returns $\mathbf{r}_i$ to the client.

The client XORs the results to obtain $\mathbf{r} = \mathbf{r}_1 \oplus \cdots \oplus \mathbf{r}_\ell = (\mathbf{v}_1 \oplus \cdots \oplus \mathbf{v}_\ell) \cdot D = \mathbf{e}_\beta \cdot D$, which is the βth block of the database, as required.

Note that this scheme is $(\ell-1)$-private; that is, no combination of $\ell-1$ or fewer servers has enough information to determine i from the information they receive from, or send to, the client. Choosing $r = s = \sqrt{n}$ yields a total communication of $2\ell\sqrt{n}$ bits to privately retrieve a block of size $\sqrt{n}$ bits.

3.3 Goldberg's PIR scheme

Chor's scheme, above, is not robust; if even one server fails to respond, the client cannot reconstruct her answer. Further, it is not Byzantine robust; if one server gives the wrong answer, then the client not only will reconstruct the wrong block, but the client will be unable to determine which server misbehaved.

Goldberg [19] modified Chor's scheme to achieve both robustness and Byzantine robustness. Rather than working over $GF(2)$ (binary arithmetic), his scheme works over a larger field $\mathbb{F}$, where each element can represent w bits (so $w = \lfloor \lg |\mathbb{F}| \rfloor$). The database D is then an $r \times s$ matrix of elements of $\mathbb{F}$. In Goldberg's simplest construction, as with Chor's scheme, each of $\ell \geq 2$ servers gets a copy of the database.

To transform this into a t-private PIR protocol, the client uses (ℓ,t) Shamir secret sharing to share the vector $\mathbf{e}_\beta \in \mathbb{F}^r$ into ℓ independent shares $(\alpha_1, \mathbf{v}_1), \ldots, (\alpha_\ell, \mathbf{v}_\ell)$. That is, the client creates r random degree-t polynomials $f_1, \ldots, f_r$ satisfying $f_j(0) = \mathbf{e}_\beta[j]$ and chooses ℓ distinct non-zero elements $\alpha_i \in \mathbb{F}$. Server i's share will be the vector $\mathbf{v}_i = \langle f_1(\alpha_i), \ldots, f_r(\alpha_i) \rangle$.

Each server then computes the product $\mathbf{r}_i = \mathbf{v}_i \cdot D = \left\langle \sum_j f_j(\alpha_i)\mathbf{w}_{j1},\ \ldots,\ \sum_j f_j(\alpha_i)\mathbf{w}_{js} \right\rangle \in \mathbb{F}^s$.

$$\begin{bmatrix} f_1(\alpha_i) & \ldots & f_r(\alpha_i) \end{bmatrix} \begin{bmatrix} \mathbf{w}_{11} & \mathbf{w}_{12} & \ldots & \mathbf{w}_{1s} \\ \mathbf{w}_{21} & \mathbf{w}_{22} & \ldots & \mathbf{w}_{2s} \\ \vdots & \vdots & & \vdots \\ \mathbf{w}_{r1} & \mathbf{w}_{r2} & \ldots & \mathbf{w}_{rs} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_j f_j(\alpha_i)\mathbf{w}_{j1} & \ldots & \sum_j f_j(\alpha_i)\mathbf{w}_{js} \end{bmatrix}$$

By the linearity property of Shamir secret sharing, since $\{(\alpha_i, \mathbf{v}_i)\}_{i=1}^{\ell}$ is a set of Shamir secret shares of $\mathbf{e}_\beta$, $\{(\alpha_i, \mathbf{r}_i)\}_{i=1}^{\ell}$ will be a set of Shamir secret shares of $\mathbf{e}_\beta \cdot D$, which is the βth block of the database. Looking at it another way, the vector $\langle \mathbf{r}_1[q], \mathbf{r}_2[q], \ldots, \mathbf{r}_\ell[q] \rangle$ is a Reed-Solomon codeword encoding the polynomial $g_q = \sum_j f_j \mathbf{w}_{jq}$, and the client wishes to compute $g_q(0)$ for each $1 \leq q \leq s$.

However, some of the servers may be down or Byzantine, so some of the shares returned by these servers may be missing or incorrect. Goldberg's scheme first optimistically assumes that all of the servers that replied gave correct responses, and uses Lagrange interpolation to attempt to reconstruct the database row (his EASYRECOVER algorithm). He bases this optimistic assumption on the fact that Byzantine servers are discovered by his scheme, which disincentivizes servers to act maliciously. If the optimism is not justified, however, his scheme then uses the Guruswami-Sudan algorithm [22] (his HARDRECOVER algorithm) to do error correction (see Section 2.3).

This scheme is t-private, and the Guruswami-Sudan algorithm can correct $v < k - \lfloor \sqrt{kt} \rfloor$ incorrect server responses. Choosing $r = s = \sqrt{n/w}$ yields a total communication of $(k+\ell)\sqrt{nw}$ bits to privately retrieve a block of size $\sqrt{nw}$ bits.

Goldberg's scheme also allows for an extension called τ-*independence* [16], in which the database itself is secret shared among the ℓ servers, so that no coalition of τ or fewer servers can learn the contents of the database. We will omit the details for ease of presentation, but our scheme extends naturally to this scenario as well.

4 Our algorithm

Our algorithm follows the same general idea as Goldberg during the client-server interaction and Shamir secret sharing. We change the way that queries are randomized and make improvements to the client-side processing to greatly improve robustness and the speed of processing.

Goldberg's block reconstruction technique uses the Guruswami-Sudan algorithm to reconstruct the block a single word at a time. However, we can achieve better error-correction bounds with the algorithm of Section 2.3.2 by considering multiple blocks simultaneously. This takes advantage of the observation that a server is either Byzantine or not; if it is not, it will give correct results for *every* query.

If the Reed-Solomon codewords the client expects to receive from the servers are $\left\langle R_1^*[q], R_2^*[q], \ldots, R_\ell^*[q] \right\rangle$ for $1 \leq q \leq s$, what it actually receives may differ because some number of servers may be down, and some further number may be Byzantine. Of the ℓ servers, it may only receive a response from k of them, and of those, v may be incorrect.

If the client receives $\langle R_1[q], R_2[q], \ldots, R_\ell[q] \rangle$ for $1 \leq q \leq s$, then:

$$\begin{bmatrix} \boxed{f_1(\alpha_1) \quad \dots \quad f_r(\alpha_1)} \\ \vdots \qquad\qquad \vdots \\ f_1(\alpha_\ell) \quad \dots \quad f_r(\alpha_\ell) \end{bmatrix} \begin{bmatrix} \mathbf{w}_{11} & \mathbf{w}_{12} & \dots & \mathbf{w}_{1s} \\ \mathbf{w}_{21} & \mathbf{w}_{22} & \dots & \mathbf{w}_{2s} \\ \vdots & \vdots & & \vdots \\ \mathbf{w}_{r1} & \mathbf{w}_{r2} & \dots & \mathbf{w}_{rs} \end{bmatrix} = \begin{bmatrix} \boxed{\begin{matrix} R_1[1] \\ \vdots \\ R_\ell[1] \end{matrix}} & \begin{matrix} R_1[2] \\ \vdots \\ R_\ell[2] \end{matrix} & \begin{matrix} \dots \\ \\ \dots \end{matrix} & \begin{matrix} R_1[s] \\ \vdots \\ R_\ell[s] \end{matrix} \end{bmatrix} = \begin{bmatrix} \boxed{\begin{matrix} g_1(\alpha_1) \\ \vdots \\ g_1(\alpha_\ell) \end{matrix}} & \begin{matrix} g_2(\alpha_1) \\ \vdots \\ g_2(\alpha_\ell) \end{matrix} & \begin{matrix} \dots \\ \\ \dots \end{matrix} & \begin{matrix} g_s(\alpha_1) \\ \vdots \\ g_s(\alpha_\ell) \end{matrix} \end{bmatrix}$$

Figure 1: Our PIR protocol illustrated. Each row of the leftmost matrix corresponds to a Shamir secret share of the database row being queried; each column of the rightmost two matrices corresponds to a Reed-Solomon codeword encoding a word of the queried database row. The client sends the ith row of the leftmost matrix to server i and expects to receive the ith row of the rightmost matrix in reply.

- For $\ell - k$ values of i, $R_i[q] = \perp$ for all q (these are the down servers)
- For at least $h = k - v$ values of i, $R_i[q] = R_i^*[q]$ for all q (these are the honest servers)
- For the remaining at most v values of i, $R_i[q] = R_i^*[q] + \Delta_{iq}$ for error terms Δ_{iq} (these are the Byzantine servers)

4.1 Randomizing queries

In order to use the algorithm of section 2.3.2, we need to ensure that the Byzantine servers produce *random* errors; that is, that the Δ_{iq} terms are randomly and independently chosen in the m codewords we supply to that algorithm. We make no a priori assumptions on the types of errors that the Byzantine servers may produce, but we will randomize the algorithm to cause any kind of spurious or malicious error to appear random.

To accomplish this, we make the following modification to Goldberg's protocol: the client chooses a uniformly random non-zero element $c_i \in_R \mathbb{F}^*$ for each server and sends the server a "blinded" query $c_i Q_i = \langle c_i f_1(\alpha_i), \dots, c_i f_r(\alpha_i) \rangle$ instead of just Q_i. When the server i returns a vector R_i', the client unblinds it by dividing by c_i to yield $R_i = c_i^{-1} R_i'$.

This ensures that if the server's response R_i' was the correct response to query $c_i Q_i$, then R_i will be the correct response to query Q_i. Further, for a Byzantine server, the error $\Delta_{iq}' = R_i'[q] - c_i R_i^*[q]$ it maliciously introduces will be randomized unpredictably by the client to $\Delta_{iq} = c_i^{-1} \Delta_{iq}' = R_i[q] - R_i^*[q]$.

Note, however, that different errors within the same server's response to a single query are *not* independently randomized; if a Byzantine server just adds a constant C to each word of the correct result $c_i R_i^*$ before returning it to the client, the client will see a result R_i that has had the constant $c_i^{-1} C$ added to each word of the correct result.

Errors from different servers, or from different queries, though, *are* independent, and it is this independence we leverage to get the linear multi-polynomial algorithm in Section 2.3.2 to work: after m queries, we will have m responses each with independently random errors, and we can use the algorithm to decode them simultaneously with high probability.

4.2 Reconstructing responses

After unblinding, the client possesses responses R_i from k servers; for ease of notation, suppose they are servers 1 through k. Each R_i is a vector $\langle R_i[1], \dots, R_i[s] \rangle$ where s is the number of words (elements of $\mathbb{F}$) in one database block. Each $\langle R_1[q], \dots, R_k[q] \rangle$ for $1 \le q \le s$ is a Reed-Solomon codeword with errors (the $\ell - k$ non-responding servers' entries having been removed) encoding a polynomial g_q; see Figure 1. The client's desired block is $\langle g_1(0), \dots, g_s(0) \rangle$.

As with Goldberg's scheme, the client first optimistically attempts to reconstruct each g_q using Lagrange interpolation on the points $\{(\alpha_1, R_1[q]), \dots, (\alpha_k, R_k[q])\}$ to see if the resulting polynomial has degree at most t. If there were no Byzantine servers, this will be successful. For any g_j for which Lagrange interpolation fails, we apply an escalating sequence of error-correction algorithms from Section 2.3 to attempt to recover g_j. Our implementation ties these together in a portfolio algorithm; see Section 5.2. If at any time, the error correction algorithm identifies a particular server as Byzantine, that server's results are discarded for all future computations.

If there is still at least one g_j which was not yet able to be reconstructed, any one such unsuccessfully decoded codeword $\langle R_1[q], \dots, R_k[q] \rangle$ is stored for later reconstruction, along with the current state of the computation. The client's requested block will not be available at this time.

The client can then do PIR requests for more blocks of the database. If it was interested in multiple blocks, it can just request those. Otherwise, it can re-request blocks it has not yet successfully decoded. Note that the properties of PIR ensure that the servers cannot tell whether a request is for a repeated block or a fresh one. Each time, the client either receives its desired block (from the Lagrange interpolation or error correcting portfolio algo-

rithms) or another codeword gets stored for later reconstruction.

When $m = \lceil \frac{v}{h-t-1} \rceil \leq v$ such codewords have been collected, we can apply the algorithm of Section 2.3.2. Since the stored codewords have independent errors, the algorithm will succeed with high probability. At that point, all m stored computations can be concluded, the m blocks will be returned to the client, and the $v < k-t-1$ Byzantine servers will be identified. The client can then avoid those servers in the future.

Note that the decoding algorithm is randomized, so there is a small chance of failure even when the client has collected the results of m queries. In this case, the client can continue to collect queries and construct new codewords until the algorithm succeeds.

Algorithm 2 summarizes the process.

Algorithm 2 Robust PIR Protocol

Goal: Client wishes to query row β from database D stored on ℓ servers.

1: Client chooses ℓ distinct non-zero elements $\alpha_1, \ldots, \alpha_\ell \in \mathbb{F}^*$.
2: Client chooses r random degree-t polynomials $f_1, \ldots, f_r \in_R \mathbb{F}[x]$ satisfying $f_j(0) = 1$ for $j = \beta$ and $f_j(0) = 0$ otherwise.
3: Client chooses ℓ random non-zero elements $c_1, \ldots, c_\ell \in_R \mathbb{F}^*$.
4: Client sends the vector

$$Q_i = \langle c_i f_1(\alpha_i), c_i f_2(\alpha_i), \ldots, c_i f_r(\alpha_i) \rangle$$

to server i.
5: Server i receives vector Q_i.
6: Server i sends the product $R'_i = Q_i \cdot D$ to client.
7: Client receives $R'_1, \ldots, R'_\ell$.
8: Client computes $R_i = c_i^{-1} R'_i$ for each i.
9: Client considers vectors $S_q = \langle R_1[q], \ldots, R_\ell[q] \rangle$ as received Reed-Solomon codewords and uses the algorithms from Section 2.3 to recover word q of row β of D.
10: If the recovery algorithm fails, postpone decoding until $m = \lceil \frac{v}{h-t-1} \rceil \leq v$ blocks have been requested (requesting blocks multiple times if necessary). Then use the algorithm from Section 2.3.2 to recover all of the blocks simultaneously.

5 Implementation and experiments

We implemented the algorithm described in this paper as an extension of Goldberg's implementation of his protocol, available as the Percy++ project on SourceForge [18]. The software is implemented in C++ using the NTL library [35].

In this paper we are concerned with the speed of the client-side block reconstruction operation in the presence of Byzantine servers. Our work does not change the server side of Goldberg's protocol in any way; to see speeds for the server-side operations, see Olumofin and Goldberg's 2011 paper [29].

5.1 Choice of underlying field

Goldberg's 2007 work used a 128-bit prime field as the field $\mathbb{F}$. Subsequent releases of Percy++, however, were able to use different fields, including prime fields of different sizes as well as $GF(2^8)$. This last field turns out to be a very efficient choice, as additions in this field can be implemented as XOR operations and multiplications are simple lookups in a 64 KB table.

Our implementation of our protocols uses C++ templates to abstract the field $\mathbb{F}$, making it very easy to work over any desired field.

5.2 Portfolio algorithms for decoding

Our implementation assembles the error correction algorithms described in Section 2.3 into a portfolio algorithm [20] to do efficient decoding. We use dynamic programming to choose an optimal sequence of decoding algorithms to try.

Each of the error correction algorithms we use is characterized by the tuple (k,t,h) with $k \geq h > t$: we wish to find a polynomial of degree at most t that passes through at least h of the k input points.

We have a few different choices of algorithm to solve this problem directly:

Berlekamp-Welch: If $h > \frac{k+t}{2}$, we can use the Berlekamp-Welch algorithm to find the unique polynomial solution, if it exists. This algorithm is quite fast, and we use it whenever it is applicable.

Guruswami-Sudan: If $h > \sqrt{kt}$, we can use the Guruswami-Sudan algorithm to find all solutions. It turns out this algorithm is very inefficient if $h^2 - kt$ is small; for the parameter sizes we care about, we avoid this algorithm if this value is less than 10.

Brute force: Lagrange interpolate each subset of $t+1$ points to form a polynomial of degree t, and see if it passes through at least h points. This works for any $h > t$, but is inefficient if $\binom{k}{t+1}$ is large.

In addition, we have three strategies to attempt to solve a particular instance with parameters (k,t,h) by recursively solving smaller instances and combining the results. Let $C(k,t,h)$ represent the expected time cost to

solve an instance of this size; we will bound this cost as a function of the costs of solving smaller instances.

Guess g incorrect points: If the client can guess that a particular point is wrong (that is, that the server that provided that point is Byzantine), then it can just throw away the point, and solve the remaining problem, with parameters $(k-1,t,h)$. In general, it might guess g points to be wrong, and solve a problem of size $(k-g,t,h)$. Since at least h of the original k points are correct, for any set of $h+g$ points, there exists a subset of g points that can be removed from the original k so that there are at least h correct points in the $k-g$ points remaining. Thus by recursively solving $\binom{h+g}{g}$ smaller instances and combining the results, we are guaranteed to find all solutions to our original problem. Thus we see that

$$C(k,t,h) \leq \min_g \binom{h+g}{g} \cdot C(k-g,t,h)$$

Guess g correct points: Conversely, the client might guess that a particular subset of g points are all correct (that is, that the servers that provided them are honest), and recursively try to find a polynomial of degree at most $t-g$ that passes through at least $h-g$ of the remaining $k-g$ points.[1] Similar to the above, we get that we need to recursively solve $\binom{k-h+g}{g}$ subproblems with parameters $(k-g,t-g,h-g)$, so we get

$$C(k,t,h) \leq \min_g \binom{k-h+g}{g} \cdot C(k-g,t-g,h-g)$$

Guess whether d points are correct or incorrect: The above strategies may not be helpful if the binomial coefficients are large. Our final strategy is to pick a fixed set of d points. For all g, and for all choices of g correct and $d-g$ incorrect points within that set, we recursively try to find polynomials of degree at most $t-g$ that pass through at least $h-g$ of the remaining $k-d$ points. As before, we get that

$$C(k,t,h) \leq \min_d \sum_g \binom{d}{g} \cdot C(k-d,t-g,h-g)$$

Given these three algorithms to directly solve the problem, and three strategies to indirectly solve it by combining solutions to smaller instances, we use dynamic programming to build a table of the best strategy to use to minimize the expected run time for inputs of each combination of parameters (k,t,h). We measure the runtimes of the direct algorithms experimentally, and compute the times for the indirect strategies. We pick the lowest result of the six, and set $C(k,t,h)$ to that value. We currently do this in a precomputation step for all $k \leq 25$ and give the PIR client access to this table.

[1] The remaining points are actually slightly modified before solving recursively. If (α^*, y^*) is guessed to be correct, then each other point (α_i, y_i) is modified to $(\alpha_i, \frac{y_i - y^*}{\alpha_i - \alpha^*})$ before recursively solving. If $f(x)$ interpolates at least $h-1$ points of the latter form, then $f(x) \cdot (x-\alpha^*) + y^*$ interpolates the corresponding $h-1$ original points and also the guessed point.

5.3 Multi-polynomial decoding

We also implemented the linear multi-polynomial decoding algorithm described in Section 2.3.2 as an extension of Percy++ using C++ and the NTL library. For the lattice reduction step, our implementation uses the lattice reduction algorithm by Mulders and Storjohann [27]. Although its theoretical runtime is not the fastest known, we chose this algorithm because of its simplicity. After the lattice reduction, the implementation then solves the resulting system of linear equations using Gaussian elimination.

As previously mentioned, if the errors are random, then there is a very low probability that this algorithm will fail. Based on experimental investigation, we conjecture the probability of failure is, to first order, $\left(\frac{1}{|\mathbb{F}|}\right)^{m(h-t-1)-v+1}$. (Recall from Section 2.3.3 that in order for the algorithm to work, $m \geq \frac{v}{h-t-1}$, or equivalently, $m(h-t-1)-v \geq 0$.) See the appendix for more details on these experiments. This probability of failure falls within the confidence intervals for all of our tests with $|\mathbb{F}| \geq 256$. We also ran tests with an extremely small field of $|\mathbb{F}| = 16$, and found that failures in that field occurred slightly (but statistically significantly) more often than our conjecture predicts. This leads us to believe that there is a missing second-order term in our conjecture, which is negligible for reasonable field sizes, but significant for tiny fields. We hope to nail down the missing term in future work.

In the cases where the linear multi-polynomial algorithm does fail, our algorithm will wait until another block is requested and then try again. This increases m by one and reduces the probability that the algorithm will fail by a factor of $|\mathbb{F}|^{h-t-1}$; therefore, since $h-t \geq 2$, the probability it will fail a second time is extremely tiny.

5.4 Measuring improvements to Percy++

In his 2007 paper, Goldberg measures the performance of his protocols using a Lenovo T60p laptop computer with a 2.16 GHz Intel dual-core CPU running Ubuntu Linux [19]. For the purposes of comparison, we have performed our measurements on a machine of the same model and similar Ubuntu Linux configuration. Goldberg reports that the implementation of

Table 1: Measuring improvements to Percy++'s client-side decoding algorithms. For these measurements we ran 100 trials using the parameters $(k,t,h) = (20,10,15)$.

Implementation	Algorithm	Field	Time
timing reported by Goldberg [19]	Guruswami-Sudan in MuPAD	128-bit prime	"several minutes"
Percy++	Guruswami-Sudan in C++	128-bit prime	9000 ± 3000 ms
Percy++	Guruswami-Sudan in C++	$GF(2^8)$	3000 ± 900 ms
this work	Cohn-Heninger in C++ with $m = 2$ blocks	128-bit prime	2.2 ± 0.9 ms
this work	Cohn-Heninger in C++ with $m = 2$ blocks	$GF(2^8)$	1.3 ± 0.4 ms

his HARDRECOVER algorithm takes "several minutes" when using the values $(k,t,h) = (20,10,15)$ [19].

Since the writing of that paper, the Percy++ software has improved. The first improvement was to implement the parts of the HARDRECOVER subroutine previously written using MuPAD in native C++. We timed HARDRECOVER 100 times using only this change, and found the running time reduced to 9000 ± 3000 ms.

The other improvement in the latest version of Percy++ is to use $GF(2^8)$ as the underlying field, rather than a 128-bit prime field. With this change, we again measured HARDRECOVER 100 times, and found the running time further reduced to 3000 ± 900 ms.

Finally, using the implementation of our algorithm described in Section 4 we further improve the running time, again tested with 100 trials using multi-polynomial decoding with just $m = 2$ blocks. With a 128-bit prime field, our algorithm completes in 2.2 ± 0.9 ms; with $GF(2^8)$, in just 1.3 ± 0.4 ms.

This is a reduction of over three orders of magnitude in client-side decoding time versus the latest software, and of over five orders of magnitude versus Goldberg's 2007 reported measurements. This comes at a cost of fetching just two blocks instead of one — something the client is likely to have done anyway. The results are summarized in Table 1.

5.5 New client-side measurements

We next outline the results of measurements taken of the implementation of our new algorithm described in Section 4. These measurements are only on the client-side decoding operations. For these measurements we used a server with a 2.40 GHz Intel dual-core CPU running Ubuntu Linux. For each case, we ran at least 100 trials, all using only a single core. We used the field $GF(2^8)$ for all experiments in this section.

To illustrate the improvements that our algorithm provides, we compare time measurements for four algorithms in Figure 2:

- the potentially exponential-time brute force decoding algorithm
- the Guruswami-Sudan list decoding algorithm from the latest release of Percy++
- the single-polynomial dynamic programming algorithm described in Section 5.2; and
- the linear multi-polynomial algorithm described in Section 5.3

Note that the data is plotted on a log scale so that the results can be easily compared, even though they span five orders of magnitude.

Observe that the Guruswami-Sudan algorithm only works when the number of Byzantine servers $v = k - h$ is less than $k - \lfloor\sqrt{kt}\rfloor$, and its running time blows up as v nears that bound. Past that bound, with more Byzantine servers, the only prior way for the client to decode the result was to use the brute-force decoding algorithm. Now, we can see that our single-polynomial dynamic programming algorithm and our multi-polynomial decoding algorithm both outperform the brute-force algorithm, often substantially. For example, in Figure 2(c) we see that for eight Byzantine servers with $(k,t) = (20,10)$, the Guruswami-Sudan algorithm is ineffective, and the brute-force algorithm takes about 10 seconds. Meanwhile, our dynamic programming algorithm takes about 1.5 seconds, and our multi-polynomial decoding algorithm takes about 6 *milliseconds*.

The multi-polynomial algorithm comes at a cost, however, of forcing the client to fetch multiple blocks. In this case, $m = \lceil \frac{v}{h-t-1} \rceil = 8$ blocks. If the client were going to fetch that many blocks anyway, there is no additional overhead to the scheme. Otherwise, the client may have to request some blocks multiple times. In the worst case, the client only wishes to fetch one block, and there are $v = k - t - 2$ Byzantine servers. In this worst case, $h = t + 2$, and the client must request its desired block $m = v = k - t - 2$ times before it will be able to decode it. Note that, even when multiple blocks are retrieved from the servers, our multi-polynomial algorithm is run only *once*, in order to distinguish the honest servers from the misbehaving ones.

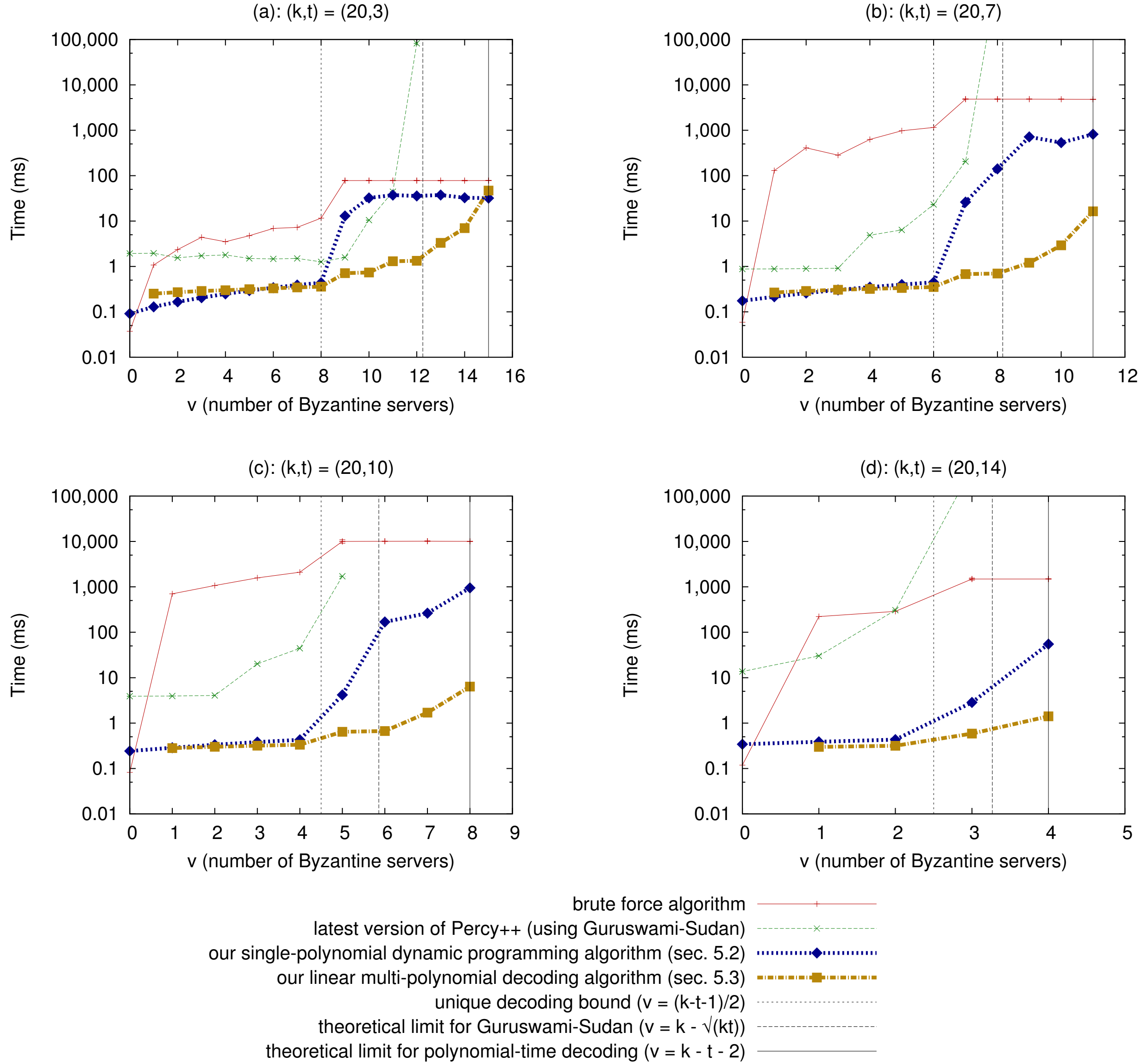

Figure 2: Timing measurements for the client-side decoding algorithms discussed in this paper for different parameters. We ran each algorithm 100 times for each choice of feasible parameters and plot the mean running times in milliseconds. Note that times are plotted on a log scale.

The three vertical lines on each plot show the unique decoding radius for Reed-Solomon codes, on the left, the theoretical bound past which the Guruswami-Sudan algorithm used in Percy++ fails, in the middle, and the theoretical bound past which efficient decoding with any algorithm is impossible, on the right.

The main results of this paper are to give two client-side decoding algorithms that outperform Guruswami-Sudan in its feasible region, and allow us to extend the range of efficient client-side decoding to the region of interest between the two vertical lines on the right. Note that the Guruswami-Sudan algorithm performs much slower in practice than the Berlekamp-Welch algorithm used by our dynamic programming portfolio algorithm within the unique decoding radius, and its running time blows up very quickly for parameters approaching its theoretical limit.

The running times of the brute force algorithm within the unique decoding radius have extremely high variance; we do not plot error bars for those timings as they obscure the entire rest of the plot. We *do* plot error bars for all other points, but they are generally too small to see.

6 Conclusions

We have improved the client side of Goldberg's 2007 Byzantine-robust information-theoretic private information retrieval protocol to use state-of-the-art decoding algorithms to improve the Byzantine robustness of the protocol to its theoretical limit. We did this using decoding algorithms that are able to take advantage of decoding information in multiple blocks of data simultaneously, observing that in practical scenarios, clients will often be interested in more than one block at a time.

We implemented our protocol and found that it is very fast in practice: *several thousand times* faster than previous protocols, and usually less than 10 ms for the parameter choices in our experiments.

Combined with fast processing on the server side [29] and scenarios in which the database servers are not in collusion [28], we can see that information-theoretic private information retrieval can be practical even in highly adversarial settings.

Acknowledgements

We thank Dan Bernstein for pointing out the connections between multi-polynomial error correction and some kinds of PIR. We thank Mark Giesbrecht and Arne Storjohann for their pointers on implementing polynomial lattice basis reduction. This material is based upon work supported by NSERC, Mprime, the National Science Foundation under Award No. DMS-1103803, and the MURI program under AFOSR Grant No. FA9550-08-1-0352. Finally, we thank the Shared Hierarchical Academic Research Computing Network (SHARCNET) and Compute/Calcul Canada for the computing cluster on which we ran the experiments in Section 5.3 and the appendix.

References

[1] C. Aguilar Melchor and P. Gaborit. A lattice-based computationally-efficient private information retrieval protocol. In *Western European Workshop on Research in Cryptology (WEWoRC2007), Bochum, Germany. Book of Abstracts*, pages 50–54, 2007.

[2] D. Asonov. Private Information Retrieval: An overview and current trends. In *ECDPvA Workshop*, 2001.

[3] A. Beimel and Y. Stahl. Robust information-theoretic private information retrieval. In *Proceedings of the 3rd International Conference on Security in Communication Networks (SCN'02)*, pages 326–341, 2003.

[4] A. Beimel and Y. Stahl. Robust information-theoretic private information retrieval. *Journal of Cryptology*, 20:295–321, 2007.

[5] E. Berlekamp and L. Welch. Error correction of algebraic block codes. US Patent Number 4,633,470, 1986.

[6] R. Carback, D. Chaum, J. Clark, J. Conway, A. Essex, P. S. Herrnson, T. Mayberry, S. Popoveniuc, R. L. Rivest, E. Shen, A. T. Sherman, and P. L. Vora. Scantegrity II Municipal Election at Takoma Park: The First E2E Binding Governmental Election with Ballot Privacy. In *19th USENIX Security Symposium*, pages 291–306, 2010.

[7] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, Feb. 1981.

[8] B. Chor and N. Gilboa. Computationally private information retrieval (extended abstract). In *29th annual ACM Symposium on Theory of Computing (STOC'97)*, pages 304–313, 1997.

[9] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Technical Report TR CS0917, Department of Computer Science, Technion, Israel, 1997.

[10] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *36th Annual IEEE Symposium on Foundations of Computer Science (FOCS'95)*, pages 41 –50, oct 1995.

[11] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45:965–981, November 1998.

[12] H. Cohn and N. Heninger. Approximate common divisors via lattices. Cryptology ePrint Archive, Report 2011/437, 2011. `http://eprint.iacr.org/`.

[13] C. Devet, I. Goldberg, and N. Heninger. Optimally Robust Private Information Retrieval. In *21st USENIX Security Symposium*, 2012.

[14] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *13th USENIX Security Symposium*, 2004.

[15] W. I. Gasarch. A survey on private information retrieval (column: Computational complexity). *Bulletin of the EATCS*, 82:72–107, 2004.

[16] Y. Gertner, S. Goldwasser, and T. Malkin. A Random Server Model for Private Information Retrieval. In *2nd International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 200–217, 1998.

[17] P. Giorgi, C.-P. Jeannerod, and G. Villard. On the complexity of polynomial matrix computations. In *2003 International Symposium on Symbolic and Algebraic Computation*, pages 135–142, 2003.

[18] I. Goldberg. Percy++ project on sourceforge. `http://percy.sourceforge.net`. Accessed February 2012.

[19] I. Goldberg. Improving the robustness of private information retrieval. In *2007 IEEE Symposium on Security and Privacy*, pages 131–148, 2007.

[20] C. Gomes and B. Selman. Algorithm Portfolios. *Artificial Intelligence*, 126(1):43–62, 2001.

[21] V. Guruswami and A. Rudra. Explicit codes achieving list decoding capacity: Error-correction with optimal redundancy. *IEEE Transactions on Information Theory*, 54(1):135–150, 2008.

[22] V. Guruswami and M. Sudan. Improved decoding of Reed-Solomon and algebraic-geometric codes. *39th Annual IEEE Symposium on Foundations of Computer Science (FOCS'98)*, pages 28–39, 1998.

[23] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *38th Annual Symposium on Foundations of Computer Science (FOCS'97)*, pages 364–373, 1997.

[24] H. W. Lenstra, A. K. Lenstra, and L. Lovász. Factoring polynomials with rational coeficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[25] S. Micali, C. Peikert, M. Sudan, and D. A. Wilson. Optimal Error Correction Against Computationally Bounded Noise. In *2nd Theory of Cryptography Conference*, pages 1–16, February 2005.

[26] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *20th USENIX Security Symposium*, pages 475–490, 2011.

[27] T. Mulders and A. Storjohann. On lattice reduction for polynomial matrices. *Journal of Symbolic Computation*, 35(4):377 – 401, 2003.

[28] F. Olumofin and I. Goldberg. Privacy-preserving queries over relational databases. In *10th International Privacy Enhancing Technologies Symposium*, pages 75–92, 2010.

[29] F. Olumofin and I. Goldberg. Revisiting the Computational Practicality of Private Information Retrieval. In *15th International Conference on Financial Cryptography and Data Security*, pages 158–172, 2011.

[30] F. Olumofin, P. Tysowski, I. Goldberg, and U. Hengartner. Achieving efficient query privacy for location based services. In *10th International Privacy Enhancing Technologies Symposium*, pages 93–110, 2010.

[31] F. Parvaresh and A. Vardy. Correcting errors beyond the Guruswami-Sudan radius in polynomial time. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pages 285–294, 2005.

[32] I. S. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics (SIAM)*, 8(2):300–304, 1960.

[33] L. Sassaman, B. Cohen, and N. Mathewson. The Pynchon Gate: a Secure Method of Pseudonymous Mail Retrieval. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society (WPES '05)*, pages 1–9, 2005.

[34] A. Shamir. How to share a secret. *Commun. ACM*, 22:612–613, November 1979.

[35] V. Shoup. NTL, a library for doing number theory. `http://www.shoup.net/ntl/`, 2005. Accessed February 2012.

[36] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2007.

[37] J. von zur Gathen. Hensel and Newton methods in valuation rings. *Math. Comp.*, 42(166):637–661, 1984.

[38] S. Yekhanin. Towards 3-query locally decodable codes of subexponential length. *J. ACM*, 55(1):1–16, 2008.

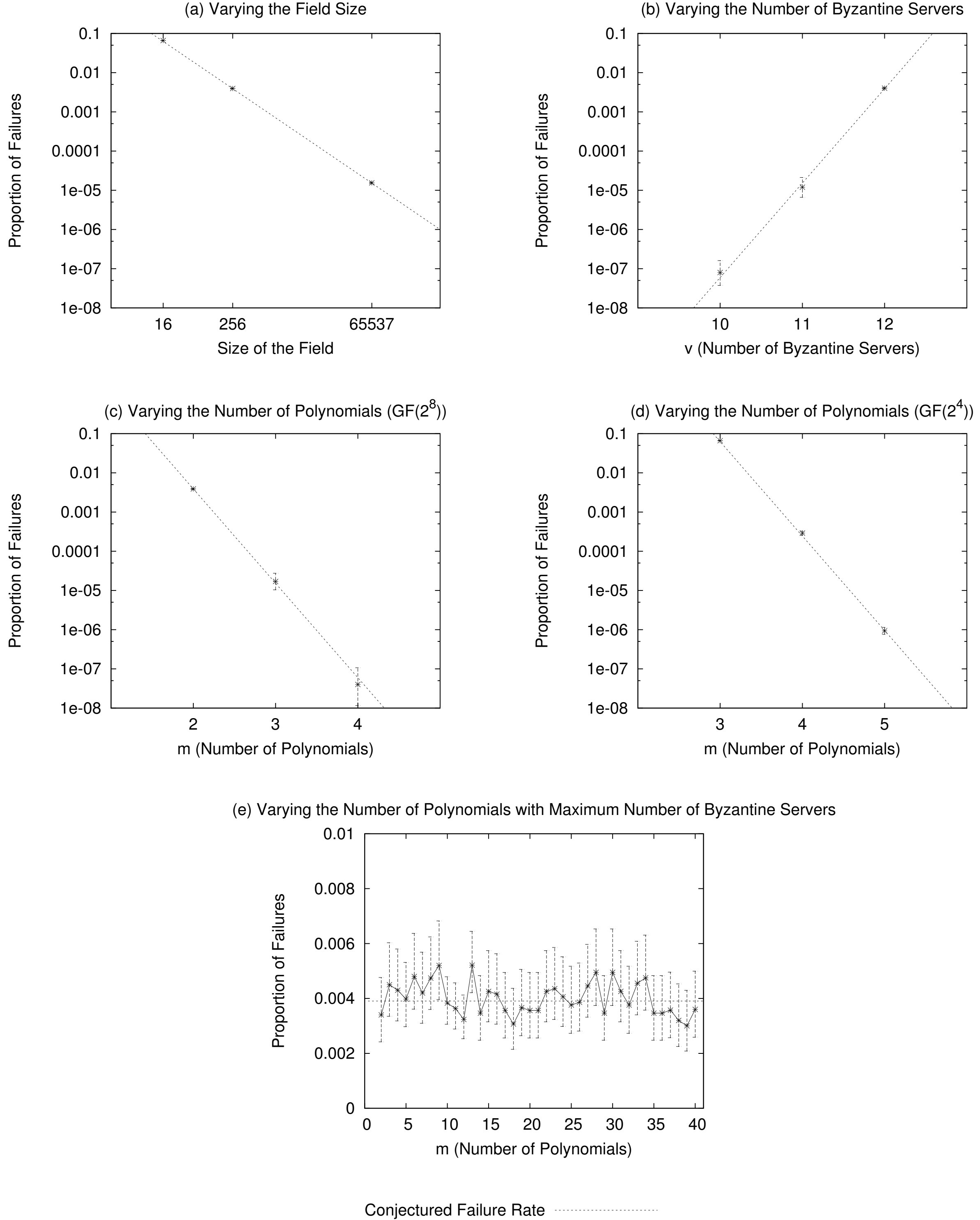

Figure 3: In Section 5.3, we conjectured that the linear multi-polynomial algorithm we present will fail with probability $\left(\frac{1}{|\mathbb{F}|}\right)^{m(h-t-1)-v+1}$. We ran several hundred million trials in order to test this conjecture. In plots (a)-(d), we varied a single parameter, keeping other parameters fixed, and plotted the observed proportion of failures for our linear multi-polynomial algorithm (black dots, with error bars at the 95% confidence interval) along with the conjectured value (dotted line). For plot (e) we varied m and used the maximum possible value of v for the given number of polynomials.

Appendix: Failure rate of Algorithm 1

The linear multi-polynomial algorithm described in Section 2.3.2 is probabilistic and may fail with some probability. We conjecture that the probability of failure is $\left(\frac{1}{|\mathbb{F}|}\right)^{m(h-t-1)-v+1}$ but do not have a proof. We ran hundreds of millions of tests, varying each of the parameters in the expression, in order to validate this conjecture experimentally. Figure 3 contains plots of the results; the raw data can be found in Table 2.

We observe that for large fields ($|\mathbb{F}| \geq 256$), our conjectured failure rate falls into the 95% confidence interval of our experimentally observed failure rate for all data points except 2 of the 39 data points in Figure 3(e). This is as expected from 95% confidence intervals. However, for a small field ($\mathbb{F} = GF(2^4)$) our conjecture appears to consistently underestimate the actual failure rate. This suggests the presence of an unknown second-order term in the failure rate; we will be exploring this in future work.

Table 2: Experimental observations of failure frequency of the multi-polynomial decoding algorithm from Section 2.3.2.

Varying $|\mathbb{F}|$:

v	t	h	m	$\lvert\mathbb{F}\rvert$	Trials	Failures	95% Confidence Interval	$\lvert\mathbb{F}\rvert^{-(m(h-t-1)-v+1)}$	In Interval
10	3	5	10	16	1,000,000	66172	$[6.56 \cdot 10^{-2}, 6.67 \cdot 10^{-2}]$	$6.25 \cdot 10^{-2}$	**No**
10	3	5	10	256	1,000,000	3956	$[3.83 \cdot 10^{-3}, 4.08 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
10	3	5	10	65537	100,000,000	1543	$[1.47 \cdot 10^{-5}, 1.62 \cdot 10^{-5}]$	$1.53 \cdot 10^{-5}$	Yes

Varying v:

v	t	h	m	$\lvert\mathbb{F}\rvert$	Trials	Failures	95% Confidence Interval	$\lvert\mathbb{F}\rvert^{-(m(h-t-1)-v+1)}$	In Interval
10	3	8	3	256	100,000,000	8	$[3.75 \cdot 10^{-8}, 16.09 \cdot 10^{-8}]$	$5.96 \cdot 10^{-8}$	Yes
11	3	8	3	256	1,000,000	12	$[0.66 \cdot 10^{-5}, 2.12 \cdot 10^{-5}]$	$1.53 \cdot 10^{-5}$	Yes
12	3	8	3	256	1,000,000	4023	$[3.90 \cdot 10^{-3}, 4.14 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes

Varying m:

v	t	h	m	$\lvert\mathbb{F}\rvert$	Trials	Failures	95% Confidence Interval	$\lvert\mathbb{F}\rvert^{-(m(h-t-1)-v+1)}$	In Interval
6	3	6	3	16	1,000,000	66087	$[6.56 \cdot 10^{-2}, 6.66 \cdot 10^{-2}]$	$6.25 \cdot 10^{-2}$	**No**
6	3	6	4	16	1,000,000	291	$[2.59 \cdot 10^{-4}, 3.26 \cdot 10^{-4}]$	$2.44 \cdot 10^{-4}$	**No**
6	3	6	5	16	100,000,000	93	$[7.58 \cdot 10^{-7}.11.40 \cdot 10^{-7}]$	$9.54 \cdot 10^{-7}$	Yes
2	3	5	2	256	1,000,000	3891	$[3.77 \cdot 10^{-3}, 4.02 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
2	3	5	3	256	1,000,000	17	$[1.03 \cdot 10^{-5}, 2.74 \cdot 10^{-3}]$	$1.53 \cdot 10^{-5}$	Yes
2	3	5	4	256	100,000,000	4	$[1.15 \cdot 10^{-8}, 10.69 \cdot 10^{-8}]$	$5.96 \cdot 10^{-8}$	Yes

Varying m and using the maximum possible value of v (in these tests, $m = \lceil \frac{v}{h-t-1} \rceil = v$):

v	t	h	m	$\lvert\mathbb{F}\rvert$	Trials	Failures	95% Confidence Interval	$\lvert\mathbb{F}\rvert^{-(m(h-t-1)-v+1)}$	In Interval
2	3	5	2	256	10000	34	$[2.42 \cdot 10^{-3}, 4.76 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
3	3	5	3	256	10000	45	$[3.35 \cdot 10^{-3}, 6.03 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
4	3	5	4	256	10000	43	$[3.18 \cdot 10^{-3}, 5.80 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
5	3	5	5	256	11564	46	$[2.97 \cdot 10^{-3}, 5.31 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
6	3	5	6	256	10000	48	$[3.61 \cdot 10^{-3}, 6.37 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
7	3	5	7	256	10000	42	$[3.09 \cdot 10^{-3}, 5.69 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
8	3	5	8	256	10754	51	$[3.60 \cdot 10^{-3}, 6.24 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
9	3	5	9	256	10000	52	$[3.96 \cdot 10^{-3}, 6.82 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	**No**
10	3	5	10	256	20100	77	$[3.06 \cdot 10^{-3}, 4.79 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
11	3	5	11	256	20100	73	$[2.88 \cdot 10^{-3}, 4.57 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
12	3	5	12	256	20100	65	$[2.53 \cdot 10^{-3}, 4.12 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
13	3	5	13	256	16308	85	$[4.21 \cdot 10^{-3}, 6.45 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	**No**
14	3	5	14	256	10100	35	$[2.48 \cdot 10^{-3}, 4.83 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
15	3	5	15	256	10100	43	$[3.15 \cdot 10^{-3}, 5.74 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
16	3	5	16	256	10100	42	$[3.06 \cdot 10^{-3}, 5.63 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
17	3	5	17	256	10100	36	$[2.56 \cdot 10^{-3}, 4.95 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
18	3	5	18	256	10100	31	$[2.15 \cdot 10^{-3}, 4.37 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
19	3	5	19	256	10100	37	$[2.64 \cdot 10^{-3}, 5.06 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
20	3	5	20	256	10100	36	$[2.56 \cdot 10^{-3}, 4.95 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
21	3	5	21	256	10100	36	$[2.56 \cdot 10^{-3}, 4.95 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
22	3	5	22	256	10100	43	$[3.15 \cdot 10^{-3}, 5.74 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
23	3	5	23	256	10100	44	$[3.23 \cdot 10^{-3}, 5.86 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
24	3	5	24	256	10100	41	$[2.98 \cdot 10^{-3}, 5.52 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
25	3	5	25	256	10100	38	$[2.73 \cdot 10^{-3}, 5.17 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
26	3	5	26	256	10100	39	$[2.81 \cdot 10^{-3}, 5.29 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
27	3	5	27	256	10100	45	$[3.32 \cdot 10^{-3}, 5.97 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
28	3	5	28	256	10100	50	$[3.74 \cdot 10^{-3}, 6.53 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
29	3	5	29	256	10100	35	$[2.48 \cdot 10^{-3}, 4.83 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
30	3	5	30	256	10100	50	$[3.74 \cdot 10^{-3}, 6.53 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
31	3	5	31	256	10100	43	$[3.15 \cdot 10^{-3}, 5.74 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
32	3	5	32	256	10100	38	$[2.73 \cdot 10^{-3}, 5.17 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
33	3	5	33	256	10100	46	$[3.40 \cdot 10^{-3}, 6.08 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
34	3	5	34	256	10100	48	$[3.57 \cdot 10^{-3}, 6.31 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
35	3	5	35	256	10100	35	$[2.48 \cdot 10^{-3}, 4.83 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
36	3	5	36	256	10100	35	$[2.48 \cdot 10^{-3}, 4.83 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
37	3	5	37	256	10071	36	$[2.57 \cdot 10^{-3}, 4.96 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
38	3	5	38	256	10000	32	$[2.25 \cdot 10^{-3}, 4.53 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
39	3	5	39	256	10000	30	$[2.09 \cdot 10^{-3}, 4.30 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes
40	3	5	40	256	10000	36	$[2.59 \cdot 10^{-3}, 4.99 \cdot 10^{-3}]$	$3.91 \cdot 10^{-3}$	Yes