# An Oblivious Password Cracking Server

Aureliano Calvo[1], Ariel Futoransky[1], and Carlos Sarraute[1,2]

[1] CoreLabs Research Center, Buenos Aires, Argentina
[2] ITBA (Instituto Tecnológico de Buenos Aires)

**Abstract.** Building a password cracking server that preserves the privacy of the queries made to the server is a problem that has not yet been solved. Such a server could acquire practical relevance in the future: for instance, the tables used to crack the passwords could be calculated, stored and hosted in cloud-computing services, and could be queried from devices with limited computing power.

In this paper we present a method to preserve the confidentiality of a password cracker—wherein the tables used to crack the passwords are stored by a third party—by combining Hellman tables and Private Information Retrieval (PIR) protocols. We provide the technical details of this method, analyze its complexity, and show the experimental results obtained with our implementation.

## 1 Introduction

Suppose that you're a hacker (or pentester) attacking a sensitive computer network, and that you've gained access to a list of password hashes. You need to retrieve the corresponding passwords to carry on with the attack... unfortunately, you don't have access to your Rainbow tables (because the computing device used to carry out the attack has limited computing power and/or memory, for example because you're using a smartphone).

This is when you need a password cracking server, that will provide access to the relevant parts of the Rainbow tables. But of course you don't want to reveal to the server (that we suppose is managed by a third party) which passwords you are trying to crack. This is the problem that we tackle in this paper: to build an "oblivious password cracking server" that contains tables of passwords and hashes, and that the users can query without revealing which pairs of passwords and hashes they are interested in.

The paper is structured as follows. In Section 2 we introduce some background on the ideas that we used. Section 3 provides an overview of the solution that we propose for this problem. In Section 4 we get into the technical details of the algorithms, in particular about the construction of the tables. Section 5 deals with the PIR protocols and provides computations of their complexity. In Section 6 we show experimental results obtained with our prototype in Python. We conclude the paper with ideas for future work.

## 2 Preliminaries

Before describing the proposed solution, we give a brief background on the ideas that we use: hash reversing tables based on time-memory trade-offs, and Private Information Retrieval (PIR) schemes to query the database server.

### 2.1 Hash Reversing Tables

A common approach in computer systems that rely on passwords for authentication is to store a cryptographic hash of the password. This approach is vulnerable to attacks based on precomputed tables for reversing the cryptographic hash function.

Martin Hellman proposed in 1980 a time-space trade-off to reverse one-way functions [7], and thus make such precomputed tables more practical. The insight of Hellman was to compute chains of hashes and passwords, and to store only the beginning and end of each chain.

Ron Rivest then proposed (in 1982) an improvement over the Hellman tables [3]. The idea was to use some (distinguished) images as chain ends in order to reduce the number of table lookups. As we will discuss in Sections 4.1 and 5.5, these tables are particularly suited for the purpose sought in this work.

In 2003, Philippe Oeschlin proposed a new improvement over the Hellman tables [11]. Instead of using a single reduction function for each chain, it would use a different one for each step in the chain. Although the Rainbow tables are faster than the Hellman tables in the general case, this is not true in the specific case of querying the database using a PIR protocol. We will discuss the details in Section 4.1.

### 2.2 Private Information Retrieval

In this work, we are interested in the case of a single database—which stores the hash reversing tables. A single-database Private Information Retrieval (PIR) scheme is a game between two players: a user and a database. The database holds some public data (for concreteness, an $n$-bit string). The user wishes to retrieve some item from the database (such as the $i$-th bit) without revealing to the database which item was queried (i.e., $i$ remains hidden) [12].

A PIR scheme usually consists of 5 steps:

1. Query generation (happens in the client)
2. Query transmission
3. Query processing (happens in the server)
4. Response transmission
5. Response decoding (happens in the client)

In the rest of the paper, we will note $O_C$ the client processing complexity (steps 1 and 5), $O_S$ the server processing complexity (step 3) and $O_T$ the transfer complexity (steps 2 and 4).

It is important to note that $O_S$ has to be at least $O(n)$ to assure that no information is leaked to the server [12]. More details about current PIR schemes are given in Section 5.1.

## 3 Our Proposed Solution

We give in this section an overview of the solution. The fundamental idea is to store Hellman tables with distinguished end-points in a series of databases accessible using a PIR protocol (see Figure 1). When a user of this system attempts to find the password corresponding to a given hash, she makes a series of PIR queries to retrieve the beginning and end of the chains corresponding to the hash being reversed.

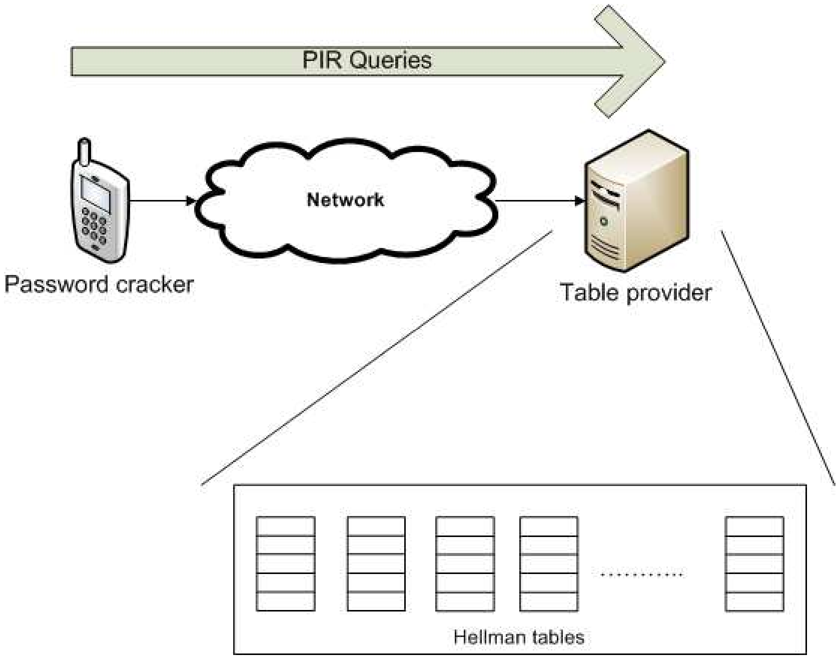

**Fig. 1.** Scheme of the proposed solution.

Let $\mathcal{P}$ be the space of passwords, $\mathcal{H}$ the space of hashes, and $H : \mathcal{P} \to \mathcal{H}$ a one-way function used to transform a plain-text $p \in \mathcal{P}$ into a hash $h \in \mathcal{H}$. We use $\mathcal{D} \subseteq \mathcal{P}$ to denote the set of passwords we are interested in, and $N$ for the number of passwords in $\mathcal{D}$. In order to quickly find in $\mathcal{D}$ the password corresponding to a given hash with probability $\alpha$, we compute and store in the server a set of $M$ Hellman tables with distinguished end-points, using $M$ different reduction functions, and whose chain length is on average $M$. The parameter $M$ depends on $\alpha$ and on the size of $\mathcal{D}$, and is in the order of $N^{1/3}$ (see Section 14 for details).

Once the tables are generated, in order to calculate the inverse of the hash $h \in \mathcal{H}$, the corresponding end-point for each reduction function $r_i$ is calculated and looked up in the tables. For each end-point found in a table, the chain is searched through looking for the preimage of $h$. If the search is successful, the corresponding password $p$ such that $h = H(p)$ is found.

Since PIR protocols are computationally expensive, our solution is to make a single query to the database for each table. A query in a PIR protocol retrieves a set of consecutive bits from the database. In order to make a single PIR query per table, the chains are stored in a closed hash table[3] sorted by the chain end-point. If a collision between the ends of two chains in the same table is found, one of the chains is discarded and another chain is calculated. We chose to discard colliding chains because we have to do exactly one PIR query per table to assure that the server does not gain information regarding the password. This was an important design decision, and is one of the contributions of this work.

To ensure that the table can be calculated in the same running order (as the regular Hellman tables with distinguished end-points), each hash table is given $\beta M$ slots. The parameter $\beta > 1$ is used to expand the hash table, in order to have enough spare buckets and avoid collisions while generating the table.

# 4 The Algorithms in Detail

## 4.1 Generating the Tables

Algorithm 1 shows the procedure used by the table provider to calculate the Hellman tables with distinguished end-points.

**Algorithm 1:** Calculate tables

```
Input: $\alpha, M$
Output: tables
tables ← empty_list()
for $i \in [0, M)$ do
    table ← new array(size = $\alpha \cdot M$, default = EMPTY_ENTRY)
    chain_count ← 0
    chain_index ← 0
    while chain_count < $M$ do
        end ← calculate_end_point(redfun(index), password_for(chain_index))
        bucket_idx ← bucket_for(end)
        if table[bucket_idx] == EMPTY_ENTRY then
            bucket_idx = (chain_index, end)
            chain_count += 1
        chain_index += 1
    tables.append(table)
return tables
```

In our implementation, the generation of the tables depends on: (i) the alphabet used by the passwords, (ii) the length of the passwords, and (iii) the desired probability $\alpha$ of cracking the passwords. Of course, the tables also depend on

[3] In a closed hash table, each bucket holds at most one entry.

the hash function $H$ to be reversed, on the reduction functions used to create the chains, and on the parameter $\beta$.

In the following subsections, we give the relevant details of the different versions of time-space trade-offs used for hash reversing. We also discuss why the Hellman tables with distinguished end-points were most suited for this application.

**Hellman Tables.** In [7] Hellman proved that by making a table of size $N^{2/3}$ a one-way function $H : \mathcal{P} \to \mathcal{H}$ can be reversed in $O(N^{2/3})$ operations where $N$ is the size of the function domain.

Let $M$ be both the number of reduction functions and the number of steps of a chain. The probability of success of finding the preimage of a hash obtained from the table domain can be estimated as $\alpha = 1 - e^{-\frac{M^3}{N}}$ [11]. This means that in order to have a success probability $\alpha$, $M$ should be computed as:

$$M = -\sqrt[3]{\ln(1-\alpha) \cdot N} \tag{1}$$

The scheme consists in chaining $M$ results by defining $M$ reduction functions $r_i : \mathcal{H} \to \mathcal{P}$. Each function $r_i$ is used to generate $M$ chains. Each chain is calculated by applying the composition of these functions $M$ times. The beginning and the end of each chain are stored in the table.

To recover the preimage of a given image $y \in \mathcal{H}$, for each $i$ apply $r_i \circ H$ at most $M$ times, until the end of a chain is found or all the posibilities are exhausted. If the end of a chain is found, the chain is generated from the beginning to retrieve the preimage of $y$. All the chains with the same reduction function $r_i$ form the table $T_i$.

**Hellman Tables with Distinguished End-Points.** The idea of [3] was to use some (distinguished) images as chain ends. For instance all images $y$ such that $y \leq K$. The parameter $K$ is chosen such that the average chain length is $M$ (from Equation 1).

This improvement reduces the number of table lookups to $M$ in the worst case (instead of $M^2$ in the original Hellman tables) and makes it easier to detect collisions between chains[4]. These are the tables and values that we use in our implementation.

**Rainbow Tables.** Instead of using a single reduction function for each chain, Rainbow tables use a different one for each step in the chain [11]. In this paper we do not use this approach because each of the PIR queries would be made to a table of size $O(M^2)$, instead of a table of size $O(M)$ (as in Hellman tables with distinguished end-points). Section 5.5 completes the comparison between the two approaches.

[4] Just compare chain ends when calculating them, and recalculate on collision. If the chain is too large, assume a cycle was formed and discard the chain

### 4.2 Password Cracker Routines

**Algorithm 2:** Crack password routine

```
   Input: hash
   Output: password or not found message
1  ends ← empty_list()
2  index ← 0
3  while index $< M$ do
4      ends.append(calculate_end_point(redfun(index),hash))
5  starts ← empty_list()
6  index ← 0
7  foreach end ∈ ends do
8      starts.append( fetch_start_PIR(end, index) )
9      index ← index +1
10 index ← 0
11 foreach start ∈ starts do
12     if start != START_NOT_FOUND then
13         solution ← find_preimage(start, redfun(index), hash)
14         if solution then
15             return solution
16     index ← index +1
17 return PREIMAGE_NOT_FOUND
```

Algorithm 2 shows the routines used by the password cracker (on the client's side). It first calculates all the posible chain buckets for all the $M$ tables on the server for the hash being reversed. Then, it looks up the buckets on the table provider (lines 7-9), which returns the end of the chain associated with this bucket and the beginning of the chain. Finally, using this information, it searches through the matching chains and finds the hash preimage.

**Algorithm 3:** fetch_start_PIR routine

```
Input: end, index
Output: fetched start
pir_db ← select_pir_db(index)
bucket_idx ← bucket_for(end)
fetched_start, fetched_end ← fetch_bucket(end, pir_db)
if fetched_end != end then
    return START_NOT_FOUND
return fetched_start
```

Algorithm 3 shows how the PIR database is used. Each query to the table provider is aimed at a different PIR database, and each database corresponds to a Hellman table. After the database is selected, the client calculates the associated bucket for the distinguished end-point. Finally, the client queries the database, which returns a pair ($fetched\_start$, $fetched\_end$). If the $fetched\_end$ is the same as the end queried, then the start is returned. If not, then the chain is not in the table queried.

## 5 Complexity Calculation

### 5.1 PIR Protocols

In this section, we provide the details of some single-database PIR schemes. Recall from Section 2.2 that $O_C$ denotes the client processing complexity, $O_S$ the server processing complexity and $O_T$ the transfer complexity.

**Naive PIR.** A very simple scheme for private information retrieval is to simply send the entire database to the client. The problem with this solution is that $O_T = O(n)$ where $n$ is the size of the database. Also $O_S = O_C = O(n)$.

**Classic PIR.** This is the first single-database PIR scheme published [4]. We will analyze the non-recursive version defined in the first part of the paper. It is based on the *Quadratic Residuosity Assumption* (QRA) [9,5,10,13]. The client sends $\sqrt{n}$ numbers to the server where all the numbers but one are squares. The server, for each row in the database, multiplies the number if and only if it is "1" and answers with the $\sqrt{n}$ multiplication results. Then the client chooses the result of interest and calculates whether it is a square. The requested bit is 0 if it is a square (and 1 if not).

Using this scheme the complexities are $O_S = n$ and $O_T = O_C = O(\sqrt{n})$.

**Fast PIR.** This scheme was published in 2007, and is based on the hidden-lattice problem [2]. The authors claim in [1] to have implemented the first practical computational PIR scheme, processing in the server 2 Gbits/s.

In this scheme, $n = O(e{\cdot}\log(e))$, where $n$ is the number of bits in the database and $e$ is the number of entries in the database. Letting $f^{-1}$ be the inverse of $f(x) = x \cdot \log(x)$, we have that $O_S = n$, $O_T = O_C = O(f^{-1}(n))$. Even though this scheme has worse client and transmission complexity, its authors claim that it works in practice better than classic PIR.

### 5.2 Password Cracker (the Client)

When a password cracker attempts to break a password, it sends $M$ queries to the table provider and for each response it iterates through the entire chain (of size $M$) looking for the preimage. Assuming that a step in the chain can be calculated in $O(1)$, the complexity for each query is $O_C + M$, where $O_C$ is the client side complexity of a single query in the PIR scheme used. The time complexity to find a preimage is thus $M \times (O_C + M)$.

We calculate below the complexity $O_C$ for the 3 PIR schemes outlined in the previous section. We denote $S$ the size of the database being queried (in our solution $S = \beta M$).

**Naive:** The complexity $O_C$ is $O(S)$.
**Classic:** The complexity $O_C$ is $O(\sqrt{S})$.
**Fast:** The complexity $O_C$ is $O(f^{-1}(S))$, where $f(x) = x \cdot \log(x)$.

As a consequence, in the three cases, the total complexity for the password cracker is dominated by $O(M^2)$. It is interesting to note that the complexity is not affected by the fact that we query the database through a PIR protocol.

### 5.3 Table Provider (the Server)

For the server, there are two stages: (i) generating the tables, and (ii) answering queries. During the second stage, when the client attempts to crack a password, it makes $M$ queries. Therefore, on the server side, the time complexity of a password crack attempt is $O(M \times O_S)$ where $O_S$ is the server processing complexity of a single query in the PIR scheme used. Recall from the previous sections that $O_S = O(\beta M) = O(M)$, so the total complexity is $O(M^2)$.

Let us analyze now the complexity of generating and storing the tables. We prove below that the table calculation is bounded in space by $\beta M^2$ and in time by $\beta' M^3$ where $\beta$ and $\beta'$ do not depend on $M$.

**Expected Number of Chain Calculations.** When the Hellman tables are calculated and stored in the table provider, our design decision was to store each Hellman table in a closed hash table, and to discard the chains when a collision is detected (see Algorithm 1). Assuming that the hash function we are trying to invert is cryptographically strong, it is safe to assume that the all the buckets are equiprobable and independent for a chain.

Say that we are calculating a table with $M$ entries in a hash table of size $k = \beta M$ (where $\beta > 1$) and that there are $i$ slots occupied with already calculated chains. The probability of a collision between the next chain to be calculated and the ones already stored is $i/k$. Therefore, the number of chains to be calculated follows a geometric probability distribution with expected value $k/k-i$. So the expected number of chain calculations for each table ($E_C$) is calculated as follows:

$$E_C = \sum_{i=1}^{M-1} \frac{k}{k-i} = k \sum_{i=1}^{M-1} \frac{1}{k-i}$$

Let $k = \beta M$, we obtain

$$E_C = \beta M \sum_{i=1}^{M-1} \frac{1}{\beta M - i}$$

Let $j = \beta M - i$, we get

$$E_C = \beta M \sum_{j=\beta M-1}^{\beta M - M + 1} \frac{1}{j} = \beta M \left[ \sum_{j=1}^{\beta M-1} \frac{1}{j} - \sum_{j=1}^{\beta M-M+1} \frac{1}{j} \right]$$

By approximating the harmonic series partial sums as $\log(n)$:

$$E_C \simeq \beta M \left(\log(\beta M - 1) - \log(\beta M - M + 1)\right)$$

Then $\forall\, M > C,\ \exists\, c > 1$ such that:

$$\begin{aligned} E_C &\leq \beta M \left(\log(\beta M) - \log(c\,(\beta - 1)M)\right) \\ &= \beta M \left(\log(\beta) + \log(M) - \log(\beta - 1) - \log(M) - \log(c)\right) \\ &= \beta M \left(\log(\beta) - \log(\beta - 1) - \log(c)\right) \end{aligned}$$

Proving that $\forall\, \beta > 1,\ \exists\, \beta'$ such that $E_C \leq \beta' M$. This means that when $M$ increases, if the ratio of unused space is kept constant in the closed hash, the time used to calculate the table increases linearly or sublinearly.

### 5.4 Transfer Complexity

The transfer complexity of a password crack attempt is $O(M \times O_T)$ where $O_T$ is the transfer complexity of a single query in the PIR scheme used. We calculate this complexity for the 3 PIR schemes outlined in Section 5.1. Again we denote $S$ the size of the database being queried (here $S = O(M)$).

**Naive:** The time complexity for a single query is $O(S)$. So the total complexity for the transfer is $O(M^2)$.

**Classic:** The time complexity for a single query is $O(\sqrt{S})$. So the total complexity for the transfer is $O(M^{3/2})$.

**Fast:** The time complexity for a single query is $O(f^{-1}(S))$, where $f(x) = x \cdot \log(x)$. So the total complexity for the transfer is $O(M \cdot f^{-1}(M))$.

### 5.5 Comparison with Other Approaches

**Brute-force.** If the password cracker brute forces the password she will not have transfer costs but she will have to iterate through the entire preimage space. So the processing costs for the password cracker will be in the order of $M^3$.

**Rainbow Tables.** This approach is much slower because in order to use a Rainbow table each query must be made against $M^2$ entry points, instead of the $M$ entry points for each table when using Hellman tables with distinguished end-points, and each PIR query resource usage depends on the number of entries in the table (see Section 5.1).

## 6 Implementation and Experimental Results

We have made a prototype that implements the solution described in this paper with the naive and classic PIR schemes. Namely we implemented hash reversing for the hash function MD5. We considered passwords using an alphabet $\mathcal{A}$ of 6 letters, and passwords lengths of 4, 5 and 6 characters. We also choose $\beta = 4$.

We focused our performance evaluation on the following parameters: (i) the length $\ell$ of the passwords ($4 \leq \ell \leq 6$), and (ii) the desired probability $\alpha$ of reversing the hashes ($0.4 \leq \alpha \leq 0.9$). By varying $\ell$ and $\alpha$, we obtain different values for $M$ (the size of the database). Recall from Section 4.1 that $M$ is computed as

$$M = -\sqrt[3]{\ln(1-\alpha) \cdot N}$$

where $N$ is the total number of passwords considered. In this case $N = |\mathcal{A}|^{\ell}$. More concretely, $N = 1296$ for $\ell = 4$, $N = 7776$ for $\ell = 5$, and $N = 46656$ for $\ell = 6$.

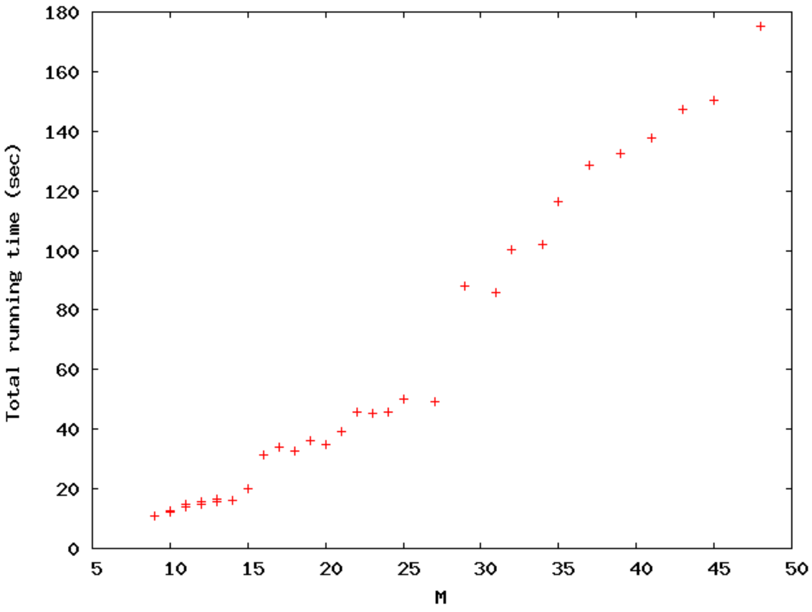

**Fig. 2.** Total running time as a function of $M$ (the size of the database).

Figure 2 shows the total runtime of the solution as a function of $M$ (each point represents the runtime to crack 100 passwords). The tests were performed on a Linux virtual machine. The guest has one CPU and 1.5 Gb of RAM. The host is also running Linux, it has 8 Gb of RAM and an Intel Xeon CPU @ 3.30GHz. The figure shows that the running time grows with $M$, but also depends on $\ell$ (the points are grouped in three "clusters" corresponding to $\ell = 4, 5, 6$). The results obtained are coherent with the estimated complexities. However we think that more extensive testing is needed to validate this approach.

We believe that our implementation still has lots of room for improvement. To begin with, the implementation is in pure Python, and thus much slower than its equivalent in C++ for example. Because of that, we are using small primes (of 20 bits) to implement classic PIR. We also think that the reduction function could be optimized. We discuss further ideas for improvement in the next section.

## 7 Conclusion and Future Steps

In this paper we tackle a problem that, as far as we know, has not been studied before. We propose a first solution for building an "oblivious password cracking server", that preserves the privacy of the queries made to the server. Even though the expected performance of this password cracker is still not sufficient to use it in real life scenarios, this scheme provides perfect privacy, and is better (in terms of complexity) than other potential approaches that we considered.

A natural future step for this project is to benefit from the recent advances in PIR research. For instance, Ian Goldberg and his group at the University of Waterloo have developed a PIR library in C++ (called Percy++), which implements protocols based on [6]. Recently, they have used it to develop PIR protocols for electronic commerce that they claim to be "practical" (presented in 2011 at the ACM CCS conference [8]). We believe that using Percy++ the results obtained here could be greatly improved.

Another promising direction is to exploit the fact that the solution proposed is inherently parallelizable. Each PIR database could run in a different host, and the client work could also be trivially parallelized. An additional improvement would be to run the password cracking server "in the cloud", with the possibility of adding processors on demand. To conclude, we hope that the first step presented here will inspire other researchers as well.

## References

1. Carlos Aguilar-Melchor. High-speed single-database pir implementation. *HotPETs*, 2008.
2. Carlos Aguilar-Melchor and Philippe Gaborit. A lattice-based computationally-efficient private information retrieval protocol. *WEWORC*, July 2007.
3. Dorothy Denning. *Cryptography and data security*, page 100. Addison Wesley, 1982.
4. Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. *Proceedings of the 38th Annu. IEEE Symp. on Foudations of Computer Science*, pages 364–373, 1997.
5. G. Brassard and C. Crepeau. Transitive transfer of confidence: A perfect zero-knowledge interactive protocol for sat and beyond. *Proc. of 27th FOCS*, pages 188–195, 1986.
6. I. Goldberg. Improving the robustness of private information retrieval. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 131–148. IEEE, 2007.
7. Martin E. Hellman. A cryptoanalytic time-memory trade off. *IEEE Transactions on Information Theory*, IT(26):401–406, 1980.
8. R. Henry, F. Olumofin, and I. Goldberg. Practical PIR for electronic commerce. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 677–690. ACM, 2011.

9. L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SICOMP*, 15:364–383, 1986.
10. M. Blum, A. De-Santis, S. Micali, and G. Persiano. Noninteractive zero-knowledge. *SIAM J. on Computing*, 20:1084–1118, 1991.
11. Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. *Advances in Cryptology - CRYPTO 2003*, 23, August 2003.
12. Rafail Ostrovsky and William E. Skeith. A survey of Single-Database PIR: Techniques and Applications. *PKC-2007 Proceedings*, 2007.
13. S. Goldwasser and S. Micali. Probabilistic encription. *Journal of Computer and systems sciences*, 28:270–299, 1984.