How to *Privately Access Remote Data*

Rafail Ostrovsky (UCLA)

**PRESENTATION AT NIST**
November 8, 2011
Contact: rafail@cs.ucla.edu

Joint works/presentation credits:
Eyal Kushilevitz, Steve Lu, William E. Skeith III.
Overview

• Motivation
• Problem Statement
• Review of PIR and ORAM
• New Results
• Conclusion
Overview

• Motivation
• Problem Statement
• Review of PIR and ORAM
• New Results
• Conclusion
Motivating Example #1
(private monitoring/reading)

I want to look up stock prices without revealing what I am looking up.
Motivating Example #2
(cloud security)
Motivating Example #3
(anti-tamper systems)
Overview

- Motivation
- Problem Statement
- Review of PIR and ORAM
- New Results
- Conclusion
Main Problem

• Encryption/Authentication protects data contents, but does not protect which physical locations are accessed.

• Access remote data without revealing the so-called “access pattern” of both reading and writing:
  – E.g.
    • Public data – stock ticker
    • Private data – cloud storage

• What do we mean by access pattern?
  – Want to hide everything about probed locations. It should look the same no matter:
    • If I ask for the same thing twice
    • If I ask for two adjacent locations
    • If I ask for random locations
  – More formally: given any two sequences of access locations, the server cannot distinguish between them
Overview

• Motivation
• Problem Statement
• Review of PIR and ORAM
• New Results
• Conclusion
One Approach: Private Information Retrieval

- Private Information Retrieval (PIR)
  - Allows client to fetch some index $i$ without revealing to the server what was retrieved
    - Information Theoretic PIR solution with replicated non-communicating servers [CGKS95]
    - [CGKS95] proved single-DB is impossible (in the info-theoretic setting)
    - Computational PIR solution with single server [KO97]
  - For now, let's talk about single-DB PIR
What is PIR?

I learned nothing about $x_i$

Remark:
For this to be a good PIR solution, the amount of data sent should be smaller than entire DB size.
Beyond PIR: distributed monitoring

• Instead of reading a single database want to monitor evolving data-sources.

• [OS95]: Searching on Streaming data.
  – (UCLA patent application, licensed to Stealth Software Technologies, Inc.)
Motivating example: “No-fly” list

- Search for classified names and aliases of suspected terrorists

- Knowledge of aliases must be kept secret
  - If not, the advantage derived from this intelligence may again become void.

- Until now, this precludes a distributed search
  - Without our technology, one must rely on an “import, then process” method
Problems with Import, then Process

• Expensive in processing
  – Processing must be done centrally

• Expensive in communication
  – Averse to dynamic data
  – Difficult to manage and synchronize data from vast and disparate sources

• Takes more information into classified setting than needed
  – sometimes can not do this (multi-agency or coalition operations)
Searching on Steaming Data

1. Classified machine: given secret search criteria, create an encrypted search and a decryption key.

2. Migrate encrypted search to multiple machines on any network (or unclassified Server Farm).

3. Every machine runs encrypted search on (local) data, writing output into a small encrypted buffer.

4. Send encrypted buffers back to a classified machine at a regular intervals (minute/hour/day).

5. Classified machine: Decrypt buffers using decryption key from step 1.
Advantages

• Attractive alternative to the import, then process paradigm

• Ideal for dynamic, distributed, streaming data

• Creates savings in communication and processing

• Enables low-latency, low-complexity monitoring
Technology at 40,000 feet

• Homomorphic encryption:
  – $E(x) \cdot E(y) = E(x+y)$

• Stealth discovery (done at UCLA)
  – $E(\text{hidden keyword}) \cdot \text{DOCUMENT} =$
    • $=E(0)$  or
    • $=E(\text{DOCUMENT})$  \textit{(only when there is a match!)}
    • Can not tell which outcome happened, just an equation

• Now can use this to “collect” only matching documents into a small encrypted buffer
Our process in detail
Step 1: Create Encrypted Search

Mohammed Atta
Hani Hanjour
Ziad Jarrah

101010101011100000
110101011000100100
101010101100110100
111110100100110100
1101011110101101
0010001110110101
100101101011101010
010101010000101110

Encrypted version of search is indistinguishable from a random distribution.
Encrypted Search:

• Provably reveals no information about search terms!

  – Therefore, it can be distributed outside of a classified environment
Step 2: Distribute Search
Step 3: Run Distributed Search

- Any willing and able parties (or a server farm) may now participate.
  - The outside participants know they are helping with a search, but remain oblivious as to what they are searching for and if there are any hits.

- Generic Interface (distributed only once) runs data “through” the encrypted search.
  - Results are collected in small encrypted buffers.
Real-Time Monitoring

• Traditional methods are unpleasant- typically complex and communication-intensive

• Constant downloads / synchronization
  – High complexity, high communication

• Waiting for batches
  – Reduces complexity, but increases latency and still involves un-necessary communication
A small encrypted flag can be frequently transmitted indicating the presence or absence of any search results. This provides a simple mechanism for real-time monitoring.
Real-Time Monitoring

The encrypted flags can be aggregated so that one small value can indicate the presence or absence of results for an entire airport, if desired.

Rather than monitoring a constant stream of thousands of names, one small value can be frequently checked on a high side.
Real-Time Monitoring

• Saves communication- only download critical data,

• Furthermore, you only download *what you were looking for, nothing else*

• Low-overhead, low-complexity method for monitoring vast data sources

• Ideal for highly dynamic data

• Ideal for situations where long knowledge latency is unacceptable
A Note on Encrypted Flags

• Encrypted flags can contain a lot, or only a little information, depending on the application

• They can give additional information, e.g. a more specific location where a hit was found and the number of hits

• If desired, it can be guaranteed to only take values of “yes” or “no”
  – Example: In coalition or multi-agency operations, one can assure that flags reveal found/not found only, and nothing else.
Step 4 and 5: upload & decrypt
Steps 4 & 5: Upload & Decrypt

• Collection of “interesting data”:
  – Transfer small buffers to a classified environment
  – Then, decrypt buffers to obtain results

• Decryption key is NEVER given to the low (i.e., unclassified) side, everything on the unclassified side is encrypted.
Design and Performance

• Designed for parallel architectures.

• Based on independently developed high-performance library for long integers and number theory.
  – In single processor, 32-bit mode, already outperforms well-established and respected libraries (e.g. NTL) on an Intel Core 2 by more than a factor of 2.
  – 64 bit mode outperforms 64 bit optimized NTL by a factor of 7 for multiplication of 1024 bit integers.
  – On a 2GHz core 2 duo in 64-bit mode can process data at 100KB/sec (small files) and 120KB/sec for large files.
  – This is about 100x faster than where we started.

• Makes use of special purpose arithmetic algorithms, ideal for the task
Another Approach: Oblivious RAM

• Oblivious RAM (ORAM)
  – Introduced by Goldreich and Ostrovsky
  – Allows client to write and read to untrusted storage encrypted data without revealing what or where it is being accessed

• We focus on this solution for the remainder of the talk
Overview

• Motivation
• Problem Statement
• Review of PIR and ORAM
• New Results
• Conclusion
Model of Oblivious RAM

• Small, trusted component
  – CPU
  – User

• Large, untrusted component
  – RAM
  – Server Farm

• **Goal:** Protect the *contents* and the *access pattern* of the small CPU from the large RAM/Cloud storage

• **PIR & ORAM Models are different:** ORAM hides *encrypted* data of CPU/User instead of reading *public data* (as in PIR).
Review: Hierarchical Solution [O90]

- Set up the Server/RAM in a hierarchy of tables
- Tables with sizes in geometric progression
  - E.g. each table is twice the size of the previous one
- Hash tables
  - Bucketed hash tables with log sized buckets
  - Cuckoo hash (need to be careful with these)
- Main property: a pair \((x,v)\) where \(x\) is a memory location and \(v\) is the contents will reside encrypted on the server and shall appear in a level \(i\) in table position \(h_i(x)\)

We drill down to the details to see how this happens.
Review: Hierarchical Solution [O90]

Reading an element

Top level is special

We scan it in its entirety

For subsequent levels $i$

compute hash $h_i(7)$

If already found, then

look up a “dummy” location instead

Since we already found memory location 7, we look up a “dummy” location

We fetched data from location 7!

I want to read memory location 7

(7, data)
Review: Hierarchical Solution [O90]
Writing an element

I want to write data to memory location (7, data)

Write to first empty location

Next Slide: We see how to update the tables as they fill up

Note: To prevent Server from distinguishing reads from writes we perform a dummy write after every read and a dummy read before every write

Encrypt

I want to write data to memory location (7, data)
Review: Hierarchical Solution [O90]

Updating the Hierarchy

- Temp Storage
- Add Dummy Elements as needed
- Compute Hash Locations
- Oblivious Sort
- Store in level

Note: It was observed in [OS97] that updates can be “smeared” over multiple read/writes to avoid long pauses during updates.
Application to Secure Computation
(Ostrovsky-Shoup Compiler [OS97])

Main Idea:
Jointly simulate ORAM, using secure circuit computation for atomic steps

Input A
Input B

Wish to compute some program $P(A,B)$ without revealing inputs

Note that many existing secure computation solutions work on circuits rather than programs.
We are able to “bootstrap” this!

This means we are able to get secure program computation with overhead proportional to that of ORAM (without unrolling program into circuit)
Overview

• Motivation
• Problem Statement
• Review
• New Results
• Conclusion
New Results

• New Insight: in Ostrovsky-Shoup complier, Alice-Bob can afford to have two non-communicating servers.

• Multi-Server Oblivious RAM [LO11]
  – Joint work with Steve Lu
  – Two (or more) non-communicating servers
    • E.g. multiple cloud services
  – $O(\log n)$ access overhead with constant client memory
    • Matches lower bound in the single-server case
  – Bypasses the expensive “oblivious sort” during updates

• Balancing Oblivious RAM [KLO11] (to appear in SODA-12)
  – Joint work with Eyal Kushilevitz and Steve Lu
  – Reduces the total overhead by balancing accesses with updates.
Multi-Server Oblivious RAM [LO11]

Main Idea

- To read a value, alternate between servers
- Let’s see how update works
Multi-Server Oblivious RAM [LO11]

Updating the levels

Move to temp via client

Compute hashes & have the server sort

Move back to other server via client

Some important additional details that we don’t discuss in this talk are needed to prevent hash overflows

CAN DO \(O(\log N)\) overhead
Balancing Oblivious RAM [KLO11]
Main Idea

- Another idea to reduce overhead
- Single server model
- Increase the size of each level to reduce the frequency of updates
  - Simply increasing the growth rate does not give us enough savings!
- Main idea: each level stores multiple hash tables
  - This reduces the frequency of updates
  - But increases the cost of reading
- How many?
  - Optimization Problem
  - For our construction turns out to be $\log(n)$ tables per level
  - Reduces total overhead down to $O(\log^2 n / \log \log n)$
Overview

• Motivation
• Problem Statement
• Review
• New Results
• Conclusion
Conclusion

• In this talk, we defined the problem of hiding the access pattern from the server

• We discussed two approaches
  – Single-DB PIR [KO97]
    • and searching in streaming model [OS05]
  – Hierarchical ORAM [O90,GO96]

• We gave additional details for the ORAM approach
  – Hierarchical Solution [O90,GO96]
  – Avoiding long pauses with “worst-case” overhead [OS97]
  – Application to secure computation [OS97]

• Described new results for ORAM [KLO12,LO11]
THANK YOU!
BACKUP: References

• [O90] Rafail Ostrovsky: Efficient Computation on Oblivious RAMs In STOC 1990: 514-523