Step-by-Step Towards a Practical Fully Encrypted Search Engine: Single Keyword, Exact Match

Gizem Çetin, Wei Dai, Yarkın Doröz and Berk Sunar

Worcester Polytechnic Institute
Presented at HEAT 2016

July 5, 2016

Search in Gentry's Dissertation

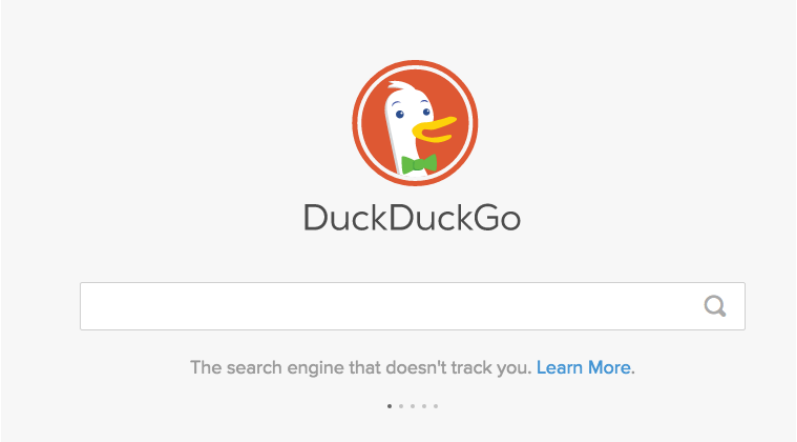
Abstract

We propose the first fully homomorphic encryption scheme, solving a central open problem in cryptography. Such a scheme allows one to compute arbitrary functions over encrypted data without the decryption key – i.e., given encryptions $E(m_1), \ldots, E(m_t)$ of $m_1, \ldots, m_t$, one can efficiently compute a compact ciphertext that encrypts $f(m_1, \ldots, m_t)$ for any efficiently computable function f. This problem was posed by Rivest et al. in 1978.

Fully homomorphic encryption has numerous applications. For example, it enables private queries to a search engine – the user submits an encrypted query and the search engine computes a succinct encrypted answer without ever looking at the query in the clear. It also enables searching on encrypted data – a user stores encrypted files on a remote file server and can later have the server retrieve only files that (when decrypted) satisfy some boolean constraint, even though the server cannot decrypt the files on its own. More broadly, fully homomorphic encryption improves the efficiency of secure multiparty computation.

Our construction begins with a somewhat homomorphic "boostrappable" encryption scheme that works when the function f is *the scheme's own decryption function*. We then show how, through recursive self-embedding, bootstrappable encryption gives fully homomorphic encryption. The construction makes use of hard problems on ideal lattices.

Who cares?



- Reached 10M searches per day
- Tightly integrated with Firefox
- See also StartPage by ixquick

How Search Engines Work

- Web crawling
- Lemmatization and indexing
- Sorting/Ranking
- Retrieval
  - Dictionary lookup
  - Autocomplete
  - Intersection between keyword lists
  - Personalization/localization

Approach

- With pre-sorted/ranked index Search becomes a PIR problem
- Leave for later:
  - Multi-keyword search (intersection)
  - Inexact search (e.g wildcards)
  - Personalization/localization
- More serious problem:
  - From PIR work we know scalability will be an issue
  - Focus first on smaller but similar problem: Autocomplete

Autocomplete Dictionary

- Three dictionaries for 1, 2 or 3 letters headers
- Comparison vs. one-hot encoding
- 20 autocomplete words per row

Headers	Words
a	amazon, abc, abc news ...
b	bank of america, best buy ...
⋮	⋮
z	zillow, zappos, zipcar ...
0	0 to 100, 0 divided by 0 ...
1	1800flowers, 1800contacts ...
2	2048, 21 day fix ...
⋮	⋮

Headers	Words					
1100001	97	109	97	122	111	...
1100010	98	97	110	107	32	...
⋮						
1111010	122	106	108	108	111	...
0110000	48	32	116	32	49	...
0110001	49	56	48	48	102	...
0110010	50	48	52	56	0	...
⋮			⋮			

Autocomplete Performance

- Vertically batched LTV implementation with DHS/cuHE libs
- Intel Xeon @ 2.9 GHz, NVIDIA GeForce GTX690
- Ubuntu 13.10., NTL 9.4.0/GMP 5.1.3.
- Vulnerable to subfield attack by Albrecht, Bai, Ducas 2016

(p, n)	#letters	With Comparison			Without Comparison		
		CPU	1 GPU	2 GPUs	CPU	1 GPU	2 GPUs
(2, 8190)	1	45 sec	267 ms	176 ms	20 sec	299 ms	168 ms
(2, 8190)	2	1.1 min	843 ms	466 ms	35 sec	304 ms	175 ms
(2, 8190)	3	1.8 min	5.42 sec	2.95 sec	58 sec	484 ms	299 ms
(257, 21504)	1	1.45 min	1.71 sec	1.00 sec	40 sec	130 ms	80 ms
(257, 21504)	2	2.2 min	2.73 sec	1.98 sec	58 sec	130 ms	80 ms
(257, 21504)	3	3.4min	8.39 sec	5.26 sec	1.3 min	130 ms	80 ms

F-NTRU (Doröz - Sunar 2016)

- **KeyGen(λ):** $\mathbb{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, $n = n(\lambda)$, message modulus $p = 2$, distributions $\chi_{\text{err}}(\lambda)$ and $\chi_{\text{key}}(\lambda)$. Sample $g, f' \in \chi_{\text{key}}$ PK: $h = 2gf^{-1}$, SK: $f = 2f' + 1$.
- **Encrypt:** Create vector of NTRU zero enc's:

$$\vec{c} = \{\text{Enc}_{\ell-1}(0), \text{Enc}_{\ell-2}(0), \ldots, \text{Enc}_0(0)\}$$

where $\text{Enc}_i(0) = hs_i + 2e_i + 0$. We call this the *ciphertext vector*. Apply BitDecomp

$$c = \text{BitDecomp}(\vec{c}^{\top})$$

Matrix C is encryption of message μ:

$$C = \text{Flatten}(I_\ell \cdot \mu + c)$$

F-NTRU

- **Decrypt:** Take the first row of C and Inverse-Bit-Decomposition to form NTRU ciphertext:

$$\text{BitDecomp}^{-1}\{c_{(0,\ell-1)}, c_{(0,\ell-2)}, \ldots, c_{(0,2)}, c_{(0,1)}, c_{(0,0)}\} = c_0.$$

  Decrypt NTRU ciphertext using secret key f, i.e. $\lfloor c_0 f \rceil \bmod 2 = \mu$.

- **Eval:** The homomorphic XOR and AND are matrix addition and multiplication operations followed by Flatten

$$C' = \text{Flatten}(C + \tilde{C}) \quad , \quad C' = \text{Flatten}(C \cdot \tilde{C}).$$

F-NTRU - One Sided Evaluation

- To decrypt C we only need the first row.
- Restrict gates to matrix on left, vector on right
- Output BW and eval. speed improved $\ell = \log(q)$ times!
- Noise growth $B_i = ||fy_i||_\infty$ with bit polynomial messages:

$$\begin{aligned} B_i &\leq \left[2n^2 B_{\text{key}} B_{\text{err}}(2^w - 1)\ell + 2n^2 B_{\text{err}}(2B_{\text{key}} + 1)(2^w - 1)\ell\right] \\ &+ \left[2n^{i+2} B_{\text{err}} B_{\text{key}} + 2n^{i+2} B_{\text{err}}(2B_{\text{key}} + 1)\right] \\ &+ \left[nB_{i-1}\right] + \left[n^{i+2}(2B_{\text{key}} + 1)\right] \end{aligned}$$

- Scalability improves greatly with single bit encryption

$$\begin{aligned} B_i &\leq \left[2n^2 B_{\text{key}} B_{\text{err}}(2^w - 1)\ell + 2n^2 B_{\text{err}}(2B_{\text{key}} + 1)(2^w - 1)\ell\right] \\ &+ \left[2nB_{\text{err}} B_{\text{key}} + 2nB_{\text{err}}(2B_{\text{key}} + 1)\right] \\ &+ \left[B_{i-1}\right] + \left[(2B_{\text{key}} + 1)\right] \end{aligned}$$

How to Encrypt the Input Keyword?

How to Encrypt the Input Keyword?

- As a string?

How to Encrypt the Input Keyword?

- As a string?
  - $[\![w]\!] \Leftrightarrow$ "depression"

How to Encrypt the Input Keyword?

- As a string?
  - $[\![w]\!] \Leftrightarrow$ "depression"
- Bit by bit?

How to Encrypt the Input Keyword?

- As a string?
  - $[\![w]\!] \Leftrightarrow$ "depression"
- Bit by bit?
  - compare every single word
  - leaks the length!

How to Encrypt the Input Keyword?

- As a string?
  - $[\![w]\!] \Leftrightarrow$ "depression"
- Bit by bit?
  - compare every single word
  - leaks the length!
- Using its index in the dictionary?

How to Encrypt the Input Keyword?

- As a string?
  - $[\![w]\!] \Leftrightarrow$ "depression"
- Bit by bit?
  - compare every single word
  - leaks the length!
- Using its index in the dictionary?

a	ℓ_1
$a1$	ℓ_2
$\vdots$	$\vdots$
$depression$	ℓ_k
$\vdots$	$\vdots$
911	ℓ_{N-1}
999	ℓ_N

How to Encrypt the Input Keyword?

- As a string?
  - $[\![w]\!] \Leftrightarrow$ "depression"
- Bit by bit?
  - compare every single word
  - leaks the length!
- Using its index in the dictionary?

0	a	ℓ_1
0	$a1$	ℓ_2
$\vdots$	$\vdots$	$\vdots$
1	*depression*	ℓ_k
$\vdots$	$\vdots$	$\vdots$
0	911	ℓ_{N-1}
0	999	ℓ_N

PIR is Key for Efficient Search

Standard Compare/Aggregate Alg. (WAHC 2014/2015)

- $\log N$ bits of the index w are encrypted.
- Using an equality check circuit $\prod_{j=1}^{\log N}(w_j \oplus i_j \oplus 1)$.
- Compare input to every single possible index value $i = 1, \cdots, N$ with bits $i_1 i_2 \cdots$.
- Bandwidth is equal to the number of bits of the index w, hence it is bounded by $s = \log N$.
- The number of multiplications will be $s - 1$ for each i.
- In total requires $N(s-1)$ ciphertext multiplications.
- Bandwidth friendly but slow!

Kushilevitz-Ostrovsky (KO) PIR

- The input index w is divided into two parts, $w = (w_1, w_2)$ where $w = w_1\sqrt{N} + w_2$
- Both w_1 and w_2 are one-hot encoded.
- Bandwidth $\sim 2^{\log(N)/2+1} = 2\sqrt{N}$ ciphertexts.
- Partition k times recursively; bandwidth $\sim kN^{1/k}$.
- Cannot use regular mul. tree for optimized KO due to one-sided eval. of F-NTRU.
- We perform k-dimensional multiplication in a serial manner.
- First, multiply along two axes: $N^{1/k} \cdot N^{1/k}$ mul's.
- Multiply results along a 3^{rd} axis; $N^{1/k} \cdot N^{2/k}$ mul's.
- After $k-1$ iterations total mul's: $N^{1/k} \sum_{i=1}^{k-1} N^{i/k}$.

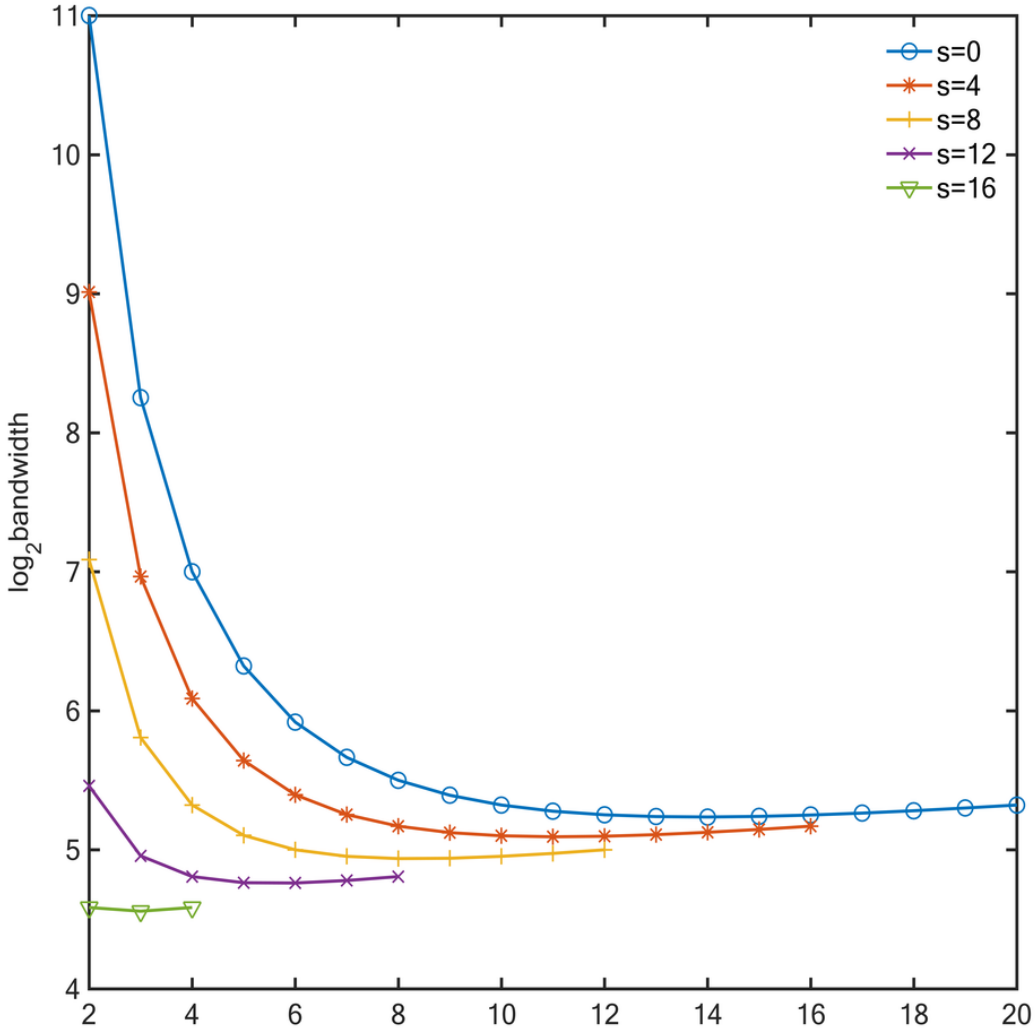
Standard/KO Hybrid PIR

- Divide input index w into two parts $w = (w_1, w_2)$ where $w = w_1 N/2^s + w_2$, i.e. w_1 is the first s bits of the index.
- Standard comparison on the first part w_1 of length s, encode w_2 of length $\log N - s$ using KO.
- Bandwidth $\sim s + k(N/2^s)^{1/k}$.
- Number of mul's: $2^s(s-1)$ and $\sum_{i=1}^{k-1} (N/2^s)^{\frac{i+1}{k}}$ (KO).
- Multiply the results of first and seconds parts; total mul's: $2^s(s-1) + \sum_{i=1}^{k-1} (N/2^s)^{\frac{i+1}{k}} + N$.

Hybrid KO Time/Bandwidth Performance

Algorithms	**Bandwidth**	**Multiplications**
Standard Comparison	$\log N$	$N(\log N - 1)$
KO Construction	$kN^{1/k}$	$\sum_{i=1}^{k-1} N^{\frac{i+1}{k}}$
Hybrid Method	$s + k(N/2^s)^{1/k}$	$2^s(s-1) + \sum_{i=1}^{k-1} (N/2^s)^{\frac{i+1}{k}} + N$

Hybrid KO Bandwidth Performance



Handling Multiple Keywords

- Need conjunction (intersection) of queries
- Define URL list $L_i = [u_{i,1}, u_{i,2}, \ldots, u_{i,t}]$ as polynomial

$$\ell_i(x) = \prod_{j=1}^{t}(u_{i,j} - x)$$

- Final output is computed as $\ell(x) = \sum_{i=1}^{N} d_i \ell_i(x)$
- To facilitate intersection take **sum** of lists, e.g. $\ell_i(x) + \ell'_i(x)$.
- Joint root will also be root of sum polynomial (Kissner and Song 2005)
- Problem: Spurious roots!

Eliminating Spurious Roots

Lemma

Let $w_1, \ldots, w_m$ be m distinct keywords with corresponding polynomials $\ell_1(x), \ldots, \ell_m(x)$ where the roots of $\ell_i(x)$ encode the top-ranked URLs bound to keyword w_i. Let $s_1, \ldots, s_m$ be m distinct primes with $s_i > T$ for all i and, for $1 \leq i \leq m$, define $r_i = \frac{1}{s_i} \prod_{h=1}^{m} s_h$. Then, for $f(x) = \sum_{i=1}^{m} r_i \ell_i(x)$,

$$\gcd\left(f(x), \prod_{u=1}^{T}(x-u)\right) = \gcd\left(\ell_1(x), \ldots, \ell_m(x)\right).$$

Handling Integer Polynomials

- To compute intersections we need to add polynomials with large coef., e.g. 50×40.
- In F-NTRU set $p = (x - 2)$ (Lauter 2014)
- Encode k-bit $\alpha = \sum_{i=0}^{k-1} \alpha_i 2^i$ as plaintext poly. $\mu = \sum_{i=0}^{k-1} \alpha_i x^i$.
- Decryption yields α; eval message poly. $\mu(x)$ at $x = 2$.
- As long as resulting poly. has degree less than n, there will be no overflow.

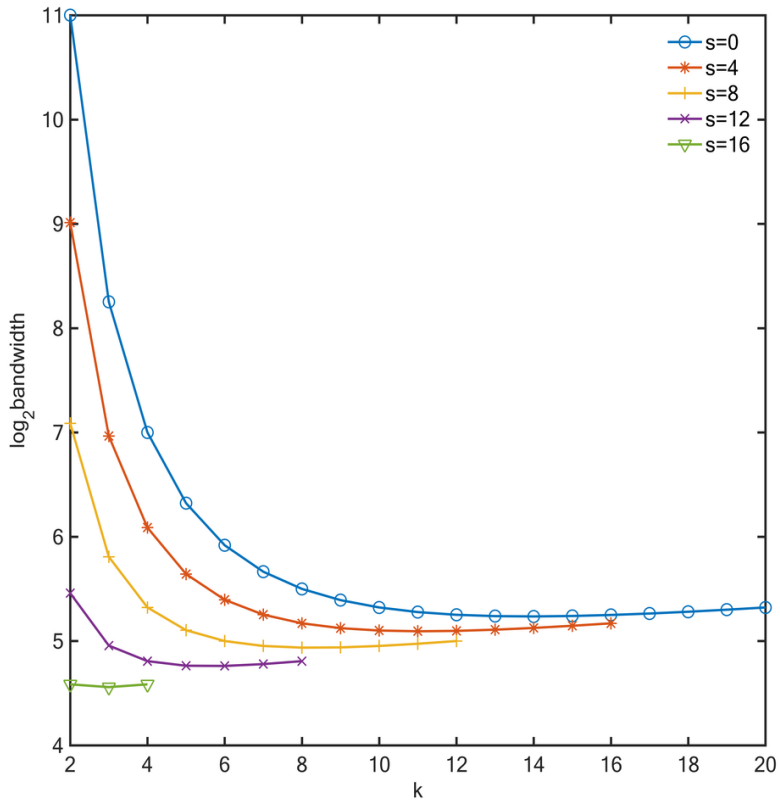
Parameter Selection

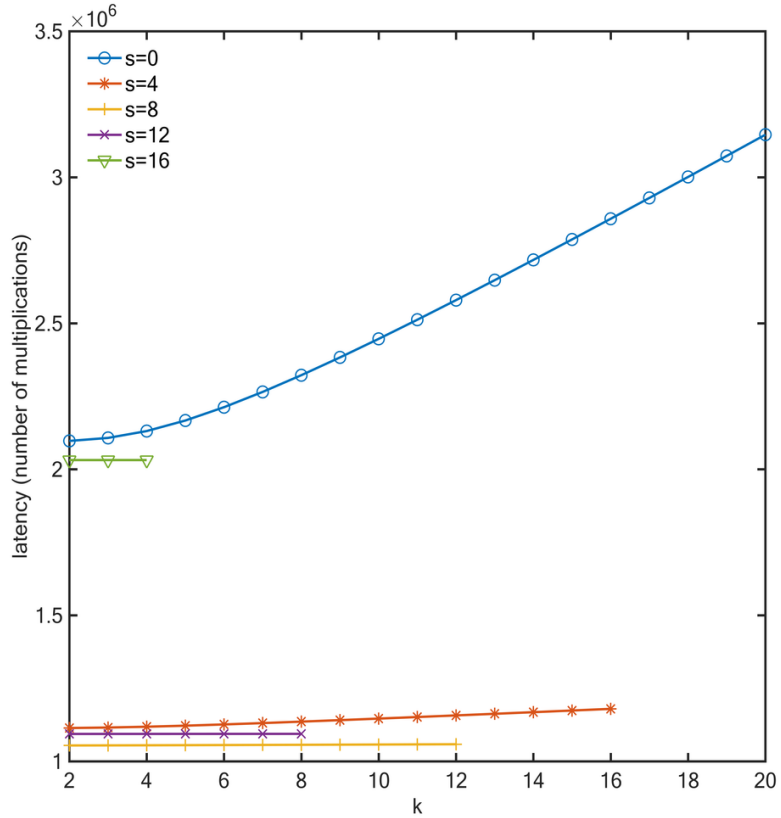
- Estimated 60 trillion $\sim 2^{46}$ web pages (most not indexed)
- URL's encoded into $u = 40$-bit strings e.g. using `bit.ly`.
- The number of input keywords is m, # URL's $t < \lfloor n/u \rfloor - m$
- Use dist. $\mathbb{D}_{\mathbb{Z}^n,\sigma_{\text{key}}}$ where $\sigma_{\text{key}} > 2n\sqrt{\log(8nq)} \cdot q^{1/2+\epsilon}$ with $\epsilon = 2^{-128}$ and $\sigma_{\text{err}} > \sqrt{\log(n)}$.
- Public key $h = g/f$, i.e. $g \in \chi_{\text{key}}$ and $f' \in \chi_{\text{key}}$ $(f = 2f + 1)$
- $\delta \leq 1.00525$ est. 128-bit sec (Lindner Peikert 2011):

$$\delta(n, q, \sigma, \epsilon) = 2^{\log^2\left(q/\sigma \cdot \sqrt{2\log(1/\epsilon)}\right)/(4n\log q)}.$$

- Concrete $n = 2039$, $q = 2^{192}$, $(m, t) = (1, 50), (2, 49), (3, 48)$

Performance Tradeoffs





Comparison of Hybrid and KO Algorithms

# Keywords	Algorithm	s	k	Bandwidth		Latency per entry (μs)
				Input	Output	
1	KO	0	2	1,152 MB	48 KB	304
1	KO	0	9	24 MB	48 KB	341
1	Hybrid	8	8	17 MB	48 KB	168
1	Hybrid	12	5	15 MB	48 KB	173
2	KO	0	9	48 MB	2.30 MB	3,544
2	Hybrid	8	8	34 MB	2.30 MB	3,198
2	Hybrid	12	5	30 MB	2.30 MB	3,207
3	Hybrid	8	8	51 MB	2.25 MB	4,722

- Database has $N = 2^{20}$ entries.
- Bandwidth: 576 KB input and 48 KB output ciphertext.
- Latency is normalized per database entry.

Conclusions

- First few step toward a blinded search engine
- Established baseline for bandwidth (MBytes) and speed (seconds) for small database
- Much slower than a real-time system
- Scalability is still a major issue
- Hardware/GPU friendly (highly parallel)
- Personalization/localization etc. still open

Thanks!