Homomorphic Encryption for Genomic Analysis

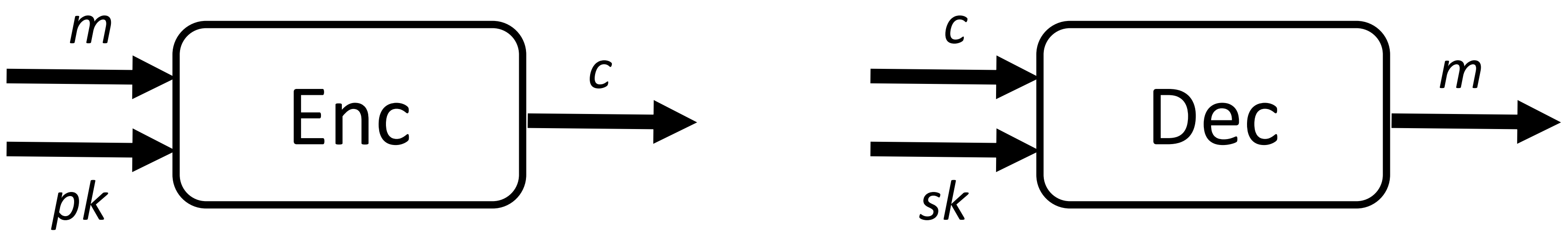
Hoon Cho (MIT) and David Wu (Stanford)

March, 2015

Homomorphic Encryption

Homomorphic encryption (HE): encryption schemes that support computation on ciphertexts

Consists of three functions:

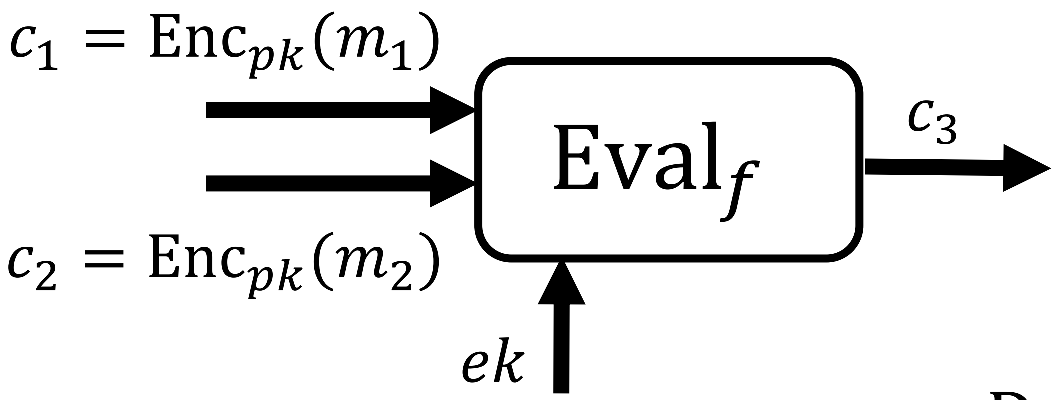


Must satisfy usual notion of semantic security

Homomorphic Encryption

Homomorphic encryption: encryption schemes that support computation on ciphertexts

Consists of three functions:



$$\mathrm{Dec}_{sk}\left(\mathrm{Eval}_f(ek, c_1, c_2)\right) = f(m_1, m_2)$$

Fully Homomorphic Encryption (FHE)

Many homomorphic encryption schemes:

- ElGamal: $f(m_0, m_1) = m_0 m_1$
- Paillier: $f(m_0, m_1) = m_0 + m_1$

Fully homomorphic encryption: homomorphic with respect to **two** operations: addition and multiplication

- [BGN05]: one multiplication, many additions (SWHE)
- [Gen09]: first FHE construction from lattices

Task 1: Computing GWAS

Case: `AA AG AA AG GG`

Control: `AG AG GA GG GG`

Genotypes for different individuals at a fixed location in the genome

allele counts

Minor Allele Frequency: $\frac{\min(n_A, n_G)}{n_A + n_G}$

χ^2-statistic: $\chi^2 = \sum \frac{(\mathrm{Obs} - \mathrm{Exp})^2}{\mathrm{Exp}}$

Observed (Obs) and expected (Exp) are functions of the different allele counts in the case and control groups

Limitations of FHE

In theory: SWHE/FHE can evaluate *arbitrary* functions

But many limitations in practice:

- Computation must be expressed as an arithmetic circuit: thus, division is hard
- Performance degrades rapidly in multiplicative depth of circuit

Striking a Balance

Minor Allele Frequency: $\frac{\min(n_A, n_G)}{n_A + n_G}$

χ^2-statistic: $\chi^2 = \sum \frac{(\text{Obs} - \text{Exp})^2}{\text{Exp}}$

Observation: allele counts are sufficient for computing MAF and χ^2

Solution: delegate *aggregation* to the cloud, client computes the statistical quantities of interest

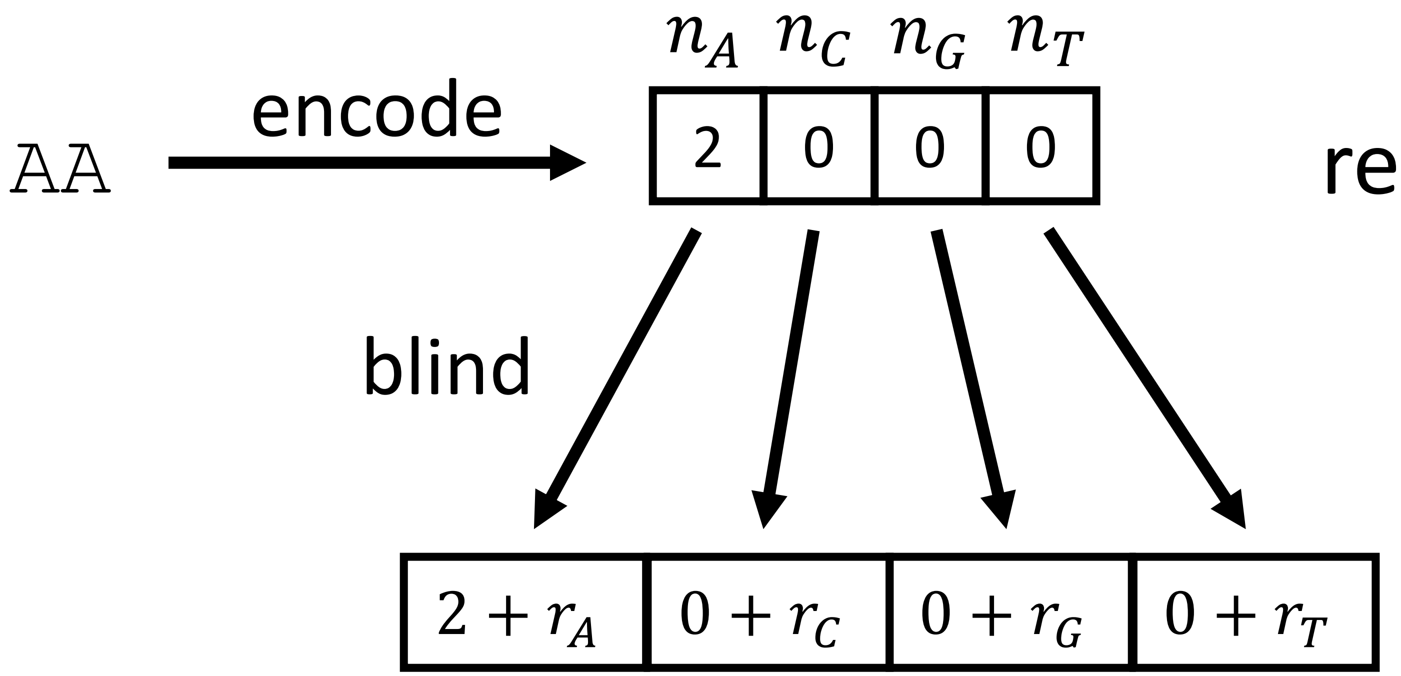
Practical Outsourcing

Solution: delegate *aggregation* to the cloud, client computes the statistical quantities of interest

Solution enables use of symmetric primitives (e.g., AES)

Symmetric primitives + arithmetic faster than public key decryption

Symmetric Encryption



each genotype represented as a vector of counts

encrypt entries by adding independent, blinding factors from $\mathbb{Z}_n$

Symmetric Encryption

AA $\longrightarrow$	$2 + r_A$	$0 + r_C$	$0 + r_G$	$0 + r_T$
AG $\longrightarrow$	$1 + r'_A$	$0 + r'_C$	$1 + r'_G$	$0 + r'_T$
Sum $\longrightarrow$	$3 + r_A + r'_A$	$0 + r_c + r'_C$	$1 + r_G + r'_G$	$0 + r_T + r'_T$

decryption: compute blinding factors and subtract

Symmetric Encryption

generate blinding factors using
$\mathrm{PRF}(k, \mathrm{tag})$

tag: `SNP id` $\|$ `group id` $\|$ `subject id`

`AA` $\longrightarrow$ | $2 + r_A$ | $0 + r_C$ | $0 + r_G$ | $0 + r_T$ |

$2 + r_A$	$0 + r_C$	$0 + r_G$	$0 + r_T$

Symmetric Encryption

Homomorphic operations consist of only **additions**

Encryption and decryption are **symmetric** primitives

Further Improvements

Client must do linear work to decrypt

- Alternative: if the data comes in batches, the client can precompute the counts per batch during encryption
- Decryption time proportional to *number of batches*

Performance

Timing (in seconds) for computing MAF + χ^2 statistics (500 subjects)

# SNPs	Encryption	Aggregation	Decryption
100	0.17	0.02	0.15
1,000	1.68	0.17	1.42
10,000	17.47	1.59	15.06
100,000	179.53	17.72	145.52

Only a few hundred lines to implement!

Task 2: Hamming Distance Computation

location of edit

edit

```
chr1:101088593: (C → T)
chr1:101265309: (C → T)
chr1:10165300:  (T → G)

        and so on…
```

```
chr1:100011666: (T → C)
chr1:101265309: (C → T)
chr1:10165300:  (T → C)

        and so on…
```

compute the Hamming distance between two sequences (represented as edits with respect to a reference genome)

Task 2: Hamming Distance Computation

```
chr1:101088593: (C → T)
chr1:101265309: (C → T)
chr1:10165300:  (T → G)

      and so on…
```

→ ATGCTTAGTGGC…

```
chr1:100011666: (T → C)
chr1:101265309: (C → T)
chr1:10165300:  (T → C)

      and so on…
```

→ ACGCTTGGTGGC…

naïve method: expand sequences, pairwise equality test

Task 2: Hamming Distance Computation

```
chr1:101088593: (C → T)
chr1:101265309: (C → T)
chr1:10165300:  (T → G)

       and so on…
```

→ `ATGCTTAGTGGC…`

sequences too long: over 3 billion base pairs in human genome

desire: protocol with performance proportional to *number of edits*

Task 2: Hamming Distance Computation

```
chr1:101088593: (C → T)
chr1:101265309: (C → T)
chr1:10165300:  (T → G)

        and so on…
```

Genome A

```
chr1:100011666: (T → C)
chr1:101265309: (C → T)
chr1:10165300:  (T → C)

        and so on…
```

Genome B

view genomes as sets of edits from reference:

$$d_H(A, B) = |A| + |B| - 2 \cdot |A \cap B|$$

Task 2: Hamming Distance Computation

Problem reduces to set intersection:

$$d_H(A, B) = |A| + |B| - 2 \cdot |A \cap B|$$

Slight caveat:

```
chr1:10165300: (T → G)
```

```
chr1:10165300: (T → C)
```

same location, different edit: contribution to Hamming distance should be 1

Task 2: Hamming Distance Computation

Formulate as two set intersection problems:

$$d_H(A, B) = |A| + |B| - |A \cap B| - \left|A^{\mathrm{loc}} \cap B^{\mathrm{loc}}\right|$$

location, edit pairs

locations only

Homomorphic Set Intersection

```
chr1:101088593: (C → T)
chr1:101265309: (C → T)
chr1:10165300:  (T → G)

        and so on…
```

```
chr1:100011666: (T → C)
chr1:101265309: (C → T)
chr1:10165300:  (T → C)

        and so on…
```

Equality function: $f(x, y) = \mathbf{1}\{x = y\}$

Simple solution: sum over pairwise equality tests

Homomorphic Set Intersection

Homomorphic evaluation of equality function:

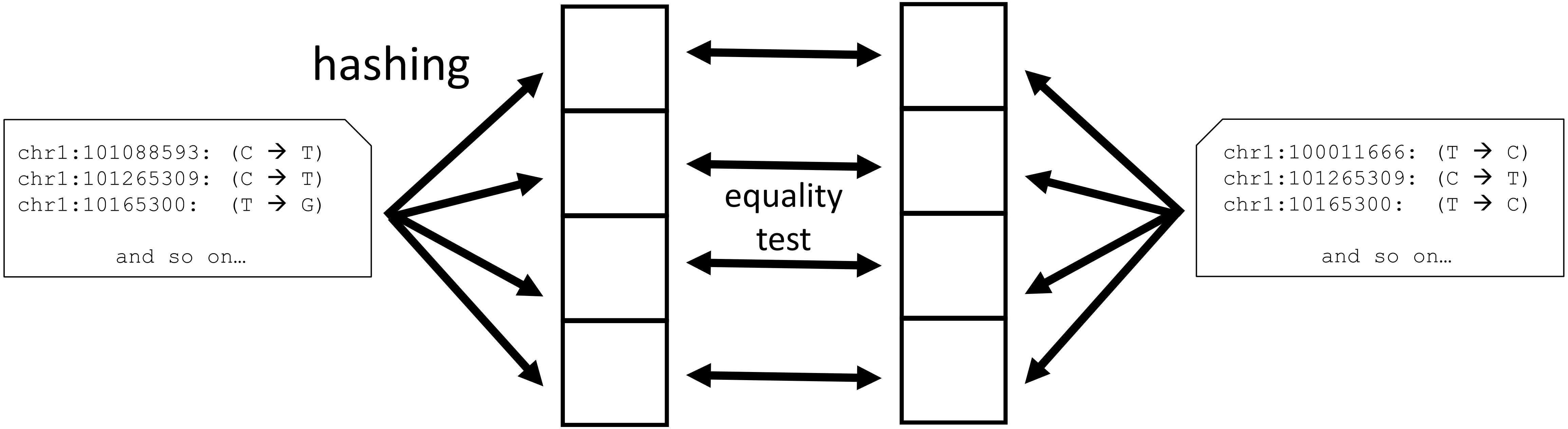
If $x, y \in \{0,1\}$,

$$f(x, y) = \mathbf{1}\{x = y\} = 1 - (x - y)^2$$

Easy to generalize to n bit integers, but requires degree $2n$ homomorphism

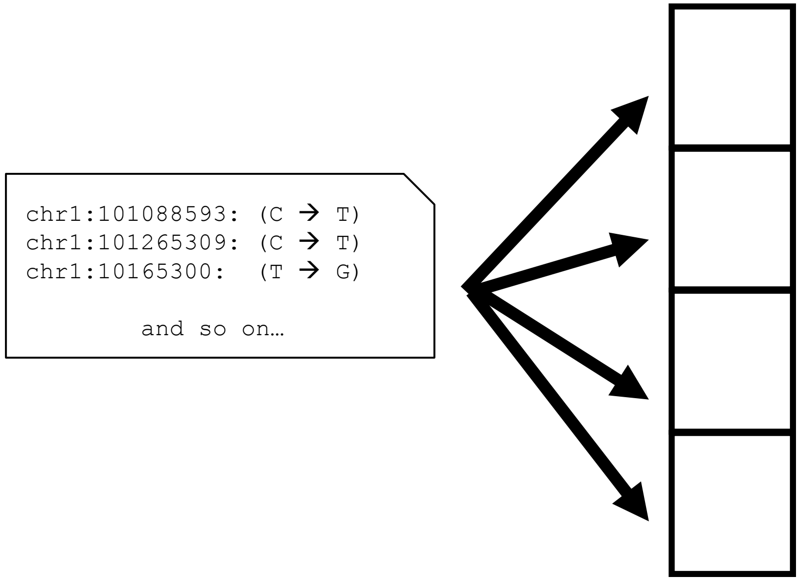
Homomorphic Set Intersection

Hashing to decrease number of pairwise comparisons



hash elements into buckets, pairwise equality test on hashed values within buckets

Homomorphic Set Intersection: Tradeoffs



More buckets → lower collision rate, possibly more ciphertexts

More bits → lower collision rate, more homomorphism for equality test

Larger buckets → less likely that bucket overflows

Tunable parameters:

- number of buckets
- bits used to represent each element in a bucket
- bucket size

Performance

Timing (in seconds) for homomorphic set intersection using HELib:

Size of Sets	Key Generation	Hashing	Encryption	Computation	Encryption
1,000	23.80	0.007	31.97	104.16	1.78
5,000	23.36	0.025	95.38	475.37	1.78
10,000	27.14	0.093	176.50	936.64	1.91

Primary drawback: key sizes + ciphertext sizes very large (several hundred MB to just over 1 GB)

Conclusions

Task 1: Most efficient solution is to compute counts – symmetric primitives suffice

Task 2: Hashing-based homomorphic set intersection can handle edit-sets with up to ten thousand elements, but with large parameter sizes