

Counting Shared Applicants Without Sharing Names

How `private-set-intersection.xyz` works

Claude

October 10, 2026

Abstract

Summer mathematics programs want to know how much their applicant pools overlap. How many distinct students apply across all of our programs? Which pairs of programs share many applicants? To how many programs does a typical student apply? But no program can hand its list of applicants to another program, or to a third party. This note describes a protocol, and a website that implements it, that answers these questions while every list of names stays on its owner’s computer. The method is the classical Diffie–Hellman *private set intersection*: each program scrambles names with a secret key, every program applies its key to every list, and because the scrambling commutes, equal names end up as equal scrambled values. Then we can count collisions, but no one can learn which names collided.

Contents

1	The question	2
2	The protocol	2
2.1	Ingredients	2
2.2	Steps	3
2.3	Why shuffle?	3
2.4	Why deduplicate?	3
3	What each party learns	3
4	Normalizing names	4
5	From distinct names to distinct people	5
6	Using the website	6
6.1	Before the session	6
6.2	Step 0: register	6
6.3	Step 1: generate a key	6
6.4	Step 2: paste and upload	7
6.5	Step 3: process everyone else’s lists	7
6.6	Results	8
7	Provenance	10

1 The question

Each participating program X has a list A_X of applicants. We would like to learn things like:

- the size of the union $|\bigcup_X A_X|$, i.e., the number of distinct applicants;
- the pairwise overlaps $|A_X \cap A_Y|$;
- the distribution of $m(s) = \#\{X : s \in A_X\}$, the number of programs to which student s applies;
- more generally, for every set S of programs, how many students applied to exactly the programs in S .

All of these quantities come from inclusion–exclusion counts. If a trusted party held every list, the computation would be trivial. The difficulty is that *nobody should hold every list*. Applicant names are personal information about minors, many programs promise in their privacy policies not to share applicant information, and it would be an uncomfortable precedent even where it is permitted.

Why not just hash the names?

A natural first idea is for every program to apply a public hash function h (say SHA-256) to each name and share the hashes. This does not protect anything. There are not that many names: anyone holding the hashed lists can hash every plausible “First Last” pair (or simply the names of their own applicants) and look the results up. Hashing with a public function is, for this purpose, the same as sharing the names.

What we need is a scrambling function that

1. no single party can evaluate on its own (so no dictionary attack is possible), and yet
2. produces the same output for the same name, no matter which program’s list it came from.

Commutative encryption provides exactly this.

2 The protocol

2.1 Ingredients

Let $\mathbb{G}$ be a cyclic group of prime order ℓ in which the Diffie–Hellman problem is hard. We use **ristretto255** [4], a prime-order group built from Curve25519, with order

$$\ell = 2^{252} + 27742317777372353535851937790883648493.$$

Elements are encoded as 32 bytes. We also need a hash function H from strings to $\mathbb{G}$; we use the standard **hash_to_ristretto255** construction of RFC 9380 [5] (SHA-512 based), with a domain-separation tag specific to this project. In the analysis, H is modeled as a random oracle: $H(x)$ behaves like a uniformly random group element.

Each program i chooses a secret key $k_i \in \{1, \dots, \ell - 1\}$ uniformly at random. To *encrypt* a group element P with key k means to compute the scalar multiple kP . (In multiplicative notation, this is $P \mapsto P^k$, exactly as in Diffie–Hellman key exchange.)

Fact (Commutativity). *For all keys a, b and all $P \in \mathbb{G}$, we have $a(bP) = (ab)P = b(aP)$. So applying keys $k_1, \dots, k_n$ in any order sends P to $(k_1 k_2 \cdots k_n)P$.*

Since ℓ is prime and $k \not\equiv 0$, the map $P \mapsto kP$ is a bijection of $\mathbb{G}$, so distinct names remain distinct at every stage.

2.2 Steps

Write $K = k_1 k_2 \cdots k_n \bmod \ell$. Each program i with (normalized, deduplicated) list A_i does the following.

1. **Upload.** Compute $\{k_i H(x) : x \in A_i\}$, shuffle it into a random order, and upload it.
2. **Process.** Repeatedly: download any list that has not yet been touched by key k_i , multiply every element by k_i , shuffle, and upload the result in its place.

When every program has touched every list, program i 's list has become

$$L_i = \{K H(x) : x \in A_i\}$$

(shuffled), *regardless of the order in which programs processed it*. Two programs' final lists contain the same element exactly when they contain the same normalized name (up to the negligible chance of a hash collision). The analysis is just counting:

$$|A_X \cap A_Y| = |L_X \cap L_Y|, \quad \left| \bigcup_X A_X \right| = \left| \bigcup_X L_X \right|, \quad \text{and so on.}$$

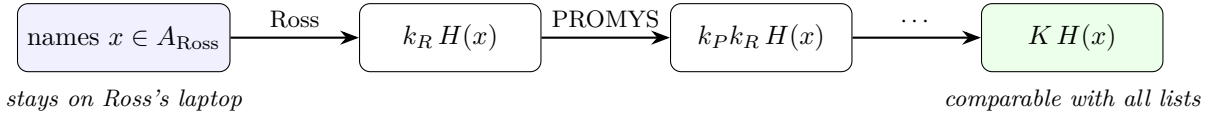


Figure 1: One program's list moving through the protocol. Every arrow is a scalar multiplication followed by a shuffle. Other lists pass through the same keys in different orders and reach the same final form.

2.3 Why shuffle?

If lists kept their order, a program that saw its own list both before and after another program's key was applied could match positions and learn which of its applicants are which encrypted values. Shuffling at every step breaks that link. In addition, the server only gives a program lists it has *not yet* processed, and never gives a program its own list back after uploading it.

2.4 Why deduplicate?

Every list is deduplicated (after normalization) before upload. If a list could contain the same encrypted value twice, a program with two applicants of the same name would recognize the pair in the final data.

3 What each party learns

The security of the scheme rests on the Decisional Diffie–Hellman assumption in $\mathbb{G}$: given P , aP , Q and R , it is infeasible to decide whether $R = aQ$. Under this assumption, and with H modeled as a random oracle, a list encrypted under keys one does not know looks like random group elements.

Party	Learns
Each program	Its own list (of course); the <i>sizes</i> of other programs' lists; nothing about the contents of other lists.
The server	Encrypted lists at every stage and each list's size and within-list duplicate counts. It never sees a name or a key: keys are generated in the browser, and only a public fingerprint k_iG is stored, to catch someone loading the wrong key file.
Organizers	The final counts: union size, pairwise overlaps, and how many encrypted names fall in each combination of programs. They can tell that <i>some</i> student applied to exactly Ross and PROMYS, but not who.

Limitations

- **Honest-but-curious.** The protocol assumes every program actually applies its key. Nothing *proves* that a program multiplied by k_i (the server checks only that it returned the right number of distinct, valid group elements). A single program that cheats is most likely to corrupt the results. Using cheating to learn about another program's applicants would take more, such as the fake second participant described next.
- **Fake participants.** Someone who registers a second, fake “program” controls two keys and could probe whether particular names appear on other lists. Registration therefore requires a join code distributed by the organizers, and the organizers can close registration once every real program has joined.
- **Small cells.** When many programs participate, some combinations will contain only one or two students. These counts are visible only to the organizers, who should avoid publishing small cells.
- **Names are an imperfect identifier.** See the next two sections.

4 Normalizing names

The protocol compares strings exactly, so the same student must produce the same string at every program. Different programs store names differently (one field or two, last-name-first, middle initials, nicknames in parentheses), so before hashing, every name is normalized in the browser:

1. remove anything in parentheses or brackets, and quoted nicknames;
2. strip accents ($\acute{e} \rightarrow e$, $\tilde{n} \rightarrow n$);
3. treat commas, tabs, and semicolons as spaces, and split into words;
4. uppercase each word and delete punctuation inside it (so hyphens and apostrophes disappear);
5. drop trailing generational suffixes (Jr, Sr, II, III, ...);
6. keep only the *first* and *last* remaining words;
7. sort those two words alphabetically and join them.

Sorting means we never need to know which word is the given name and which is the family name, so “Lee, Justin” and “Justin Lee” agree.

Pasted	Normalized key
Jim Jordan	JIM JORDAN
Joe Jameson	JAMESON JOE
Lee, Justin	JUSTIN LEE
Justin Lee (1 of 2)	JUSTIN LEE
Anne-Marie Dubois	ANNEMARIE DUBOIS
José Núñez	JOSE NUNEZ
Matthew A. Husen Jr.	HUSEN MATTHEW

This deliberately introduces a few false matches. Two different students named Justin Lee become one key, and so do “Jim Jordan” and “Jordan Jim”. It also misses some true matches (“Matt Smith” versus “Matthew Smith”). We ask programs to paste legal names, which removes most of the second problem, and we *measure* the first problem, as described next.

5 From distinct names to distinct people

Each program’s own list is already a list of distinct *people*. So when the browser normalizes a list, every repeated key is a genuine name collision among distinct people. The browser records the *multiplicity spectrum*: for each $m \geq 1$, the number of keys that occur exactly m times in that list. Only these counts are uploaded.

Let p_j be the fraction of applicants whose normalized name is j , and let

$$C_2 = \sum_j p_j^2,$$

the probability that two randomly chosen applicants share a normalized name. From a list of n people whose spectrum has s_m keys of multiplicity m , the number of colliding pairs is $\sum_m s_m \binom{m}{2}$ out of $\binom{n}{2}$ pairs. Pooling over programs gives the estimate

$$\hat{C}_2 = \frac{\sum_{\text{programs}} \sum_m s_m \binom{m}{2}}{\sum_{\text{programs}} \binom{n}{2}}.$$

Now suppose there are N distinct people in total, and U distinct normalized names in the union. Name j is absent from the union only if none of the N people has it, so

$$\mathbb{E}[U] = \sum_j (1 - (1 - p_j)^N) \approx \sum_j \left(N p_j - \binom{N}{2} p_j^2 \right) = N - \binom{N}{2} C_2,$$

using $\sum_j p_j = 1$. We therefore estimate the number of people by solving

$$N = U + \binom{N}{2} \hat{C}_2$$

by fixed-point iteration. In practice C_2 is on the order of 10^{-5} or smaller, so the correction is small. That small size is itself useful to know: it tells us that “matching by name” is accurate enough for the questions we are asking.

The same reasoning, applied with higher moments $C_r = \sum_j p_j^r$, could refine the estimate and produce corrections for the pairwise counts. We have not needed this.

6 Using the website

All of the cryptography runs in the participant’s browser, in a background thread (Web Worker). The browser code is not minified, so anyone can read it with “view source”. The server (Node.js, Express, SQLite) stores only encrypted lists, counts, and a public key fingerprint per program.

6.1 Before the session

Each program should prepare a plain-text list of applicant **legal names**, one per line, first and last name in either order. Programs should leave duplicate names in, because the page counts them, and need not clean up markers like “(1 of 2)”. Optionally, programs can prepare two lists, one of completed applications and one of all applicants. These are analyzed separately.

6.2 Step 0: register

One person per program registers with the program’s name, a password, and the join code from the organizers (Figure 2).

private-set-intersection.xyz
Counting shared applicants across summer math programs — without sharing names.

Log in	Register your program
Program name 	Program name e.g. MathILy
Password 	Your name / email (optional, for the organizers)
Log in	Choose a password
	Repeat password
	Join code (from the organizers)
	Register

Figure 2: Logging in or registering a program.

6.3 Step 1: generate a key

The page generates the program’s secret key, keeps it in the browser, and downloads a backup copy (*psi-key-program.json*). Keep this file until the exercise is over. If you switch computers or browsers, load the backup and the page will verify that it matches. Do not send the key to anyone, including the organizers.

6.4 Step 2: paste and upload

Paste the list. The page immediately shows how many distinct keys the list contains, how each name is normalized, and which rows look suspicious, such as email addresses, digits, or single words (Figure 3). Fix anything that is wrong, then click *Encrypt and upload*. The names never leave the computer; only the encrypted points are uploaded.

2 Upload your applicant lists

Completed applications

Paste legal names, one per line. Leave duplicate names in — the page counts them. Order of first/last name does not matter. Nothing you paste here is sent anywhere; only encrypted values are uploaded.

Gideon Zhou
Daniel Gallo
Kira Rahman
Justin Lee (1 of 2)
Lee, Justin
Anne-Marie Dubois
José Núñez
jlee2009@example.com
Madonna

267 names → 266 distinct after normalizing. (1 name appearing 2×.)

▼ How names are normalized (first 5 rows)

Line	You pasted	Becomes	Note
1	Gideon Zhou	GIDEON ZHOU	
2	Daniel Gallo	DANIEL GALLO	
3	Kira Rahman	KIRA RAHMAN	
4	Justin Lee (1 of 2)	JUSTIN LEE	
5	Lee, Justin	JUSTIN LEE	

▼ 2 rows look unusual — please check

Line	You pasted	Becomes	Note
8	jlee2009@example.com	JLEE2009EXAMPLECOM	looks like an email address; contains digits; only one name
10	Madonna	MADONNA	only one name

Figure 3: Pasting a list. The two flagged rows are an email address and a single name, both probably mistakes.

6.5 Step 3: process everyone else's lists

Click *Start processing* and leave the tab open. The page repeatedly asks the server for a list that still needs this program's key, applies the key, and uploads the result. There are no fixed rounds: lists flow to whichever programs are ready. If two programs grab the same list at the same moment, the server accepts only one result, and the other program simply moves on and retries later. The progress board (Figure 4) shows which program has processed which list, so it is easy to see who is still working.

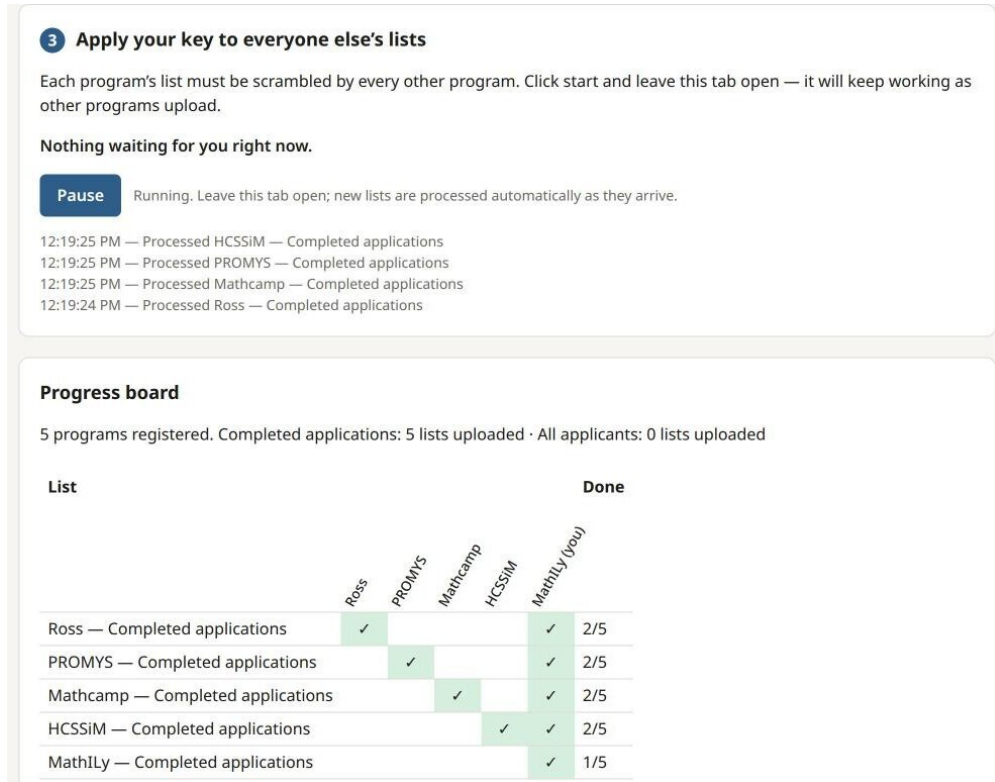


Figure 4: Partway through: MathILy has applied its key to every list, but the other programs have not yet processed MathILy's list or one another's.

6.6 Results

When every list has been processed by every program, the organizer page shows the results (Figures 5 and 6). These include:

- the union size and the estimated number of distinct people;
- for each program, how many applicants appear on no other list;
- the pairwise overlap matrix;
- the distribution of the number of programs per applicant;
- counts for every exact combination of programs.

The organizers can also download a de-identified CSV with one row per encrypted name and one 0/1 column per program, for further analysis.

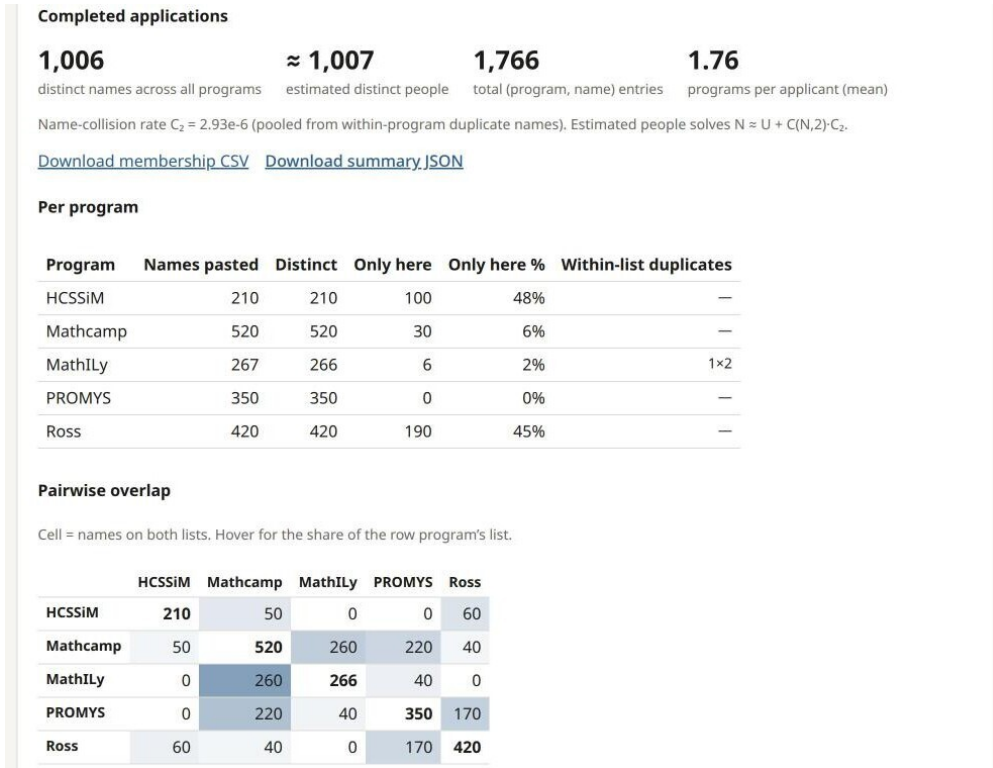


Figure 5: Organizer view of the results, using synthetic data for five programs.

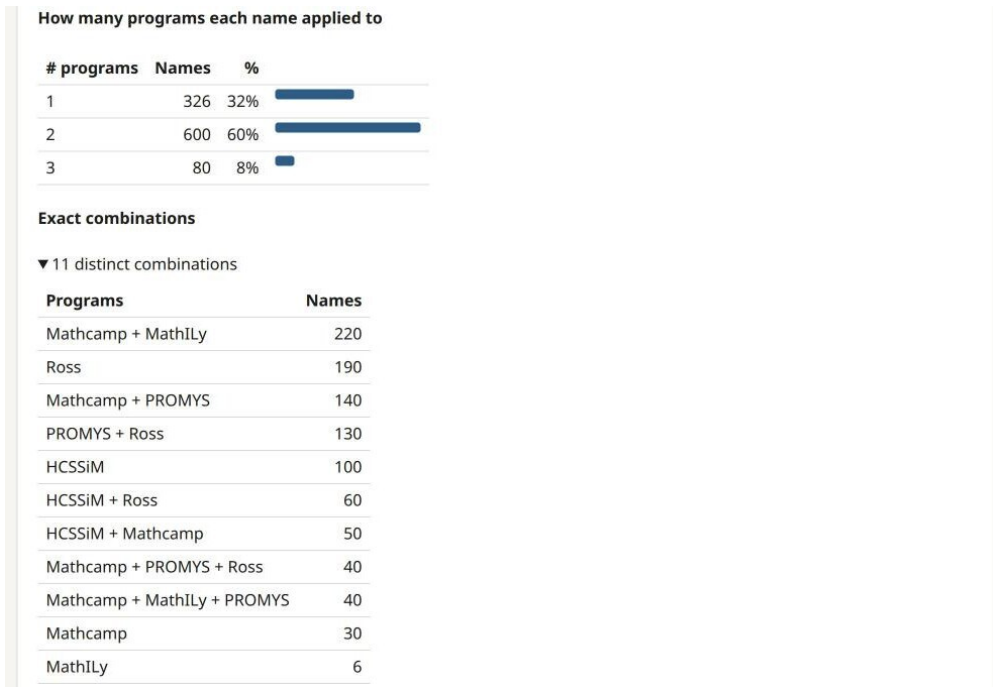


Figure 6: How many programs each (encrypted) name applied to, and the exact combinations.

7 Provenance

We did not invent this cryptography. Commutative-encryption set intersection goes back to Meadows [1] and to Huberman, Franklin, and Hogg [2], and it is a standard example of private set intersection [3]. The group (ristretto255) and the hash-to-group construction are IETF standards [4, 5], and the implementation uses the audited @noble/curves library [6]. The source code for the website is at <https://github.com/kisonecat/private-set-intersection.xyz>.

References

- [1] C. Meadows, “A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party,” *IEEE Symposium on Security and Privacy*, 1986.
- [2] B. A. Huberman, M. Franklin, and T. Hogg, “Enhancing privacy and trust in electronic communities,” *ACM Conference on Electronic Commerce*, 1999.
- [3] B. Pinkas, T. Schneider, and M. Zohner, “Scalable private set intersection based on OT extension,” *ACM Transactions on Privacy and Security* 21(2), 2018. See also https://en.wikipedia.org/wiki/Private_set_intersection.
- [4] H. de Valence, J. Grigg, M. Hamburg, I. Lovecruft, G. Tankersley, and F. Valsorda, “The ristretto255 and decaf448 Groups,” RFC 9496, 2023.
- [5] A. Faz-Hernández, S. Scott, N. Sullivan, R. S. Wahby, and C. A. Wood, “Hashing to Elliptic Curves,” RFC 9380, 2023.
- [6] P. Miller, noble-curves, <https://github.com/paulmillr/noble-curves>.