Lessons Learned: Building a Privacy-Preserving Entity Resolution Adaptation of PPJoin using End-to-End Homomorphic Encryption

Tanmay Ghai*
Information Sciences Institute
Marina del Rey, USA
tghai@isi.edu

Yixiang Yao*
University of Southern California
Los Angeles, USA
yixiangy@usc.edu

Srivatsan Ravi
University of Southern California
Los Angeles, USA
srivatsr@usc.edu

***Abstract*—Entity resolution is the task of disambiguating records that refer to the same entity in the real world. In this work, we explore adapting one of the most efficient and accurate Jaccard-based entity resolution algorithms - PPJoin, to the private domain via end-to-end homomorphic encryption. Towards this, we present our precise adaptation: HE-PPJoin that details certain subtle data structure modifications and algorithmic additions needed for correctness and privacy. We implement HE-PPJoin by extending the PALISADE (now merged with OpenFHE) open-source, homomorphic encryption library and perform experiments to analyze its accuracy and incurred overhead. Furthermore, we directly compare HE-PPJoin against P4Join, an existing privacy-preserving variant of PPJoin, which uses hashing for raw content obfuscation (encryption), by demonstrating a rigorous analysis of the efficiency, accuracy, and privacy properties achieved by our adaptation as well as a characterization of those same attributes in P4Join. In building and designing HE-PPJoin, we faced numerous challenges that required making tradeoffs and analyzing possible alternatives. We have thus summarized and detailed all the lessons we have learned, presented throughout the paper, intended as motivating building blocks for future work in this direction.**

1. Introduction

The *entity resolution* problem involves finding pairs across datasets that belong to different owners which refer to the same entity in the real world. For example, consider two datasets (e.g. D_1, D_2) with distinct collections of records (e.g. $\{r_{11}, r_{12}, r_{13}\}$, $\{r_{21}, r_{22}\}$) containing representative information about *The Avengers* as detailed in Figure 1. The task here is to determine which pairs between D_1 and D_2 correspond to the same superheroes. As shown in the figure, (r_{12}, r_{21}) both refer to "Tony Stark" (commonly known as Iron Man), while (r_{13}, r_{22}) correspond to "Stephen Strange" (or Dr. Strange).

There exist many traditional entity resolution algorithms that, in many cases, directly compare each pair of records among datasets resulting in quadratic time complexity. PPJoin [33] algorithm is one such example that seeks to be more *efficient* by pruning the record comparisons as part of the search space.

In addition to efficiency, one common concern for applications of entity resolution where the datasets may contain sensitive information (e.g. medicine, financial institutions) is that of *privacy*. In such cases, the goal is to perform entity resolution while not revealing any identifiable or sensitive data content to any data owners or to an adversary; solving entity resolution given this additional setting is known as *privacy-preserving entity resolution*. P4Join [30] is an existing privacy-preserving variant of PPJoin established to provide privacy utilizing encrypted bit vectors. While P4Join attempts to hide the raw content of data it operates over, it remains susceptible to possible attacks and mis-configurations that can reveal sensitive information to adversaries. Thus, we contend that the security it provides is *not* sufficient in the real-world.

We, therefore, investigate a *provably* secure adaptation of PPJoin into the privacy-preserving setting using homomorphic encryption [2]. Towards this, we implement new data structure operations and perform algorithmic modifications to establish our adapted version: HE-PPJoin; evaluating over real-world data and settings given an adversary model, we rigorously compare our adaptation against P4Join for efficiency, accuracy, and security.

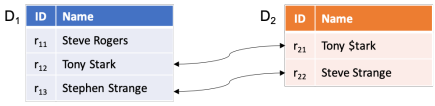

Figure 1. An example of entity resolution. Pairs (r_{12}, r_{21}) and (r_{13}, r_{22}) refer to the same entities in the real world.

Summary of our contributions In this paper, we propose an end-to-end homomorphic adaptation of PPJoin. We provide the complete pseudocode that outlines the required algorithmic modifications and analyze the performance and accuracy of the adapted algorithm using synthetic data. To do this, we *invent* an n-party protocol that maintains PPJoin's correctness and efficiency properties, while also guaranteeing *no* leakage of fatally sensitive information; we additionally ensure that there is *no* accuracy loss. For comparison, we develop a P4Join Python implementation and provide a rigorous comparison against it, detailing possible security flaws and running time vs. accuracy trade-offs. Lastly, we summarize and detail all lessons learned presented as key findings for motivating extended work in this domain.

2. Motivation and Preliminaries

2.1. Problem Definition

The **entity resolution** (ER) problem can be described as an algorithmic set $P = (D_1 \dots D_n, M)$ where $D_1 \dots D_n$ are datasets consisting of a collection of *unique*

*. These authors contributed equally to this work

records, from *different* data owners or parties $P_1 \dots P_n$. M denotes the matching record pairs (r_i, r_j denoting record ids) between any two datasets amongst the n parties, that is, $M = \{(r_i, r_j) \,|\, r_i = r_j; r_i \in D_k, r_j \in D_l\}$, where $r_i = r_j$ indicates that r_i and r_j refer to the same entity in the real world, and D_k, D_l are any two datasets $\in \{D_1 \dots D_n\}$. Normally, $n \in [1, \infty)$. When $n = 1$, the ER task is *within* one dataset, called **de-duplication**. The M in this case is, $M = \{(r_i, r_j) \,|\, r_i = r_j; r_i, r_j \in D\}$, where both r_i and r_j come from the same dataset D.

The **privacy-preserving entity resolution** (PPER) problem extends ER by requiring the original data to be obfuscated in such a way that it is not exposed to entities part of the computation model or to an adversary. In general, there are many methods that can be used to do this (i.e. hashing), but for our purposes, we opt for encryption. Keeping the same structure as with ER, here, $P_1 \dots P_n$ encrypt their records and send $Enc(D_1) \dots Enc(D_n)$ to a computation host to determine M', which is analogous to M, but over encrypted data: $M' = \{(r_i, r_j) \,|\, r_i = r_j, r_i \in Enc(D_k), r_j \in Enc(D_l)\}$. Additionally, the probability $P_1 \dots P_n$ learn any information outside of their dataset D_i or the matching M' is negligible, and the probability that a computation host learns anything outside of M', including anything in or about $\{D_1 \dots D_n\}$, is negligible amongst the presence of adversaries.

2.2. Jaccard Similarity

To determine whether two records (r_i, r_j) form a pair in the real world, string similarity metrics can be used to score how similar they are based on their representative information. Among various string similarity metrics, Jaccard similarity is well-known and widely used. Given two sets of tokens x and y, the Jaccard similarity between them can be described as the size of the intersection divided by the size of the union, that is, $Sim_{Jaccard}(x, y) = \frac{|x \cap y|}{|x \cup y|} = \frac{|x \cap y|}{|x| + |y| - |x \cap y|}$.

The Jaccard similarity is computed over n-gram tokenized raw input, that is, $Sim_{Jaccard}(x, y) \geq t$ where $x = tokenize(r_i)$, $y = tokenize(r_j)$. Each (r_i, r_j) is called a *candidate pair*, which is considered to be a *true* pair iff. the final similarity score is above a certain threshold. The total number of candidate pairs is the Cartesian product between sizes of D_k and D_l, i.e. $|D_k| \times |D_l|$, which is known as a *full comparison*.

Exhausting quadratic full comparison for finding true pairs is impractical in the real world for it is *not* scalable. For such reasons, pruning a full comparison by removing pairs that are unlikely to be part of M, *before* comparison, with relatively low-budget methods is far more feasible. Blocking [26], [35] is one category of such methods. However, the benefit of blocking also comes with an inherent side-effect: true pairs might not be thoroughly recognized, and it requires a non-trivial amount of tuning in practice to be effective. Unlike blocking, PPJoin, is an algorithm specifically designed to exploit features of Jaccard similarity in order to avoid a true full comparison. Though its complexity is still quadratic in the worst case, in *many* instances it is able to successfully and efficiently filter out candidate pairs that are not likely to be in M *without* sacrificing true pairs part of M [19].

2.3. PPJoin

PPJoin (Position Prefix Join) [33] is an extended implementation of Jaccard similarity that addresses the Cartesian product shortcoming of the naive implementation by drastically reducing the number of candidate pairs that must be considered, leading to much-improved efficiency. The intuition behind PPJoin is to avoid comparison between record pairs that have huge length differences or an overlap of tokens that is below some threshold.

PPJoin utilizes three filtering methods derived from the basic Jaccard similarity. Suppose the input sets are x and y, *length filtering* is directly converted from $Sim_{Jaccard}(x, y) \geq t$ so that $t \cdot |x| \leq |y|$. Next, fragment overlaps among similar records [8] defines *prefix filtering* where the prefix $pref(x)$ of a record x is computed as $|pref(x)| = \lceil (1-t) \cdot |x| \rceil + 1$, and $(|x| - \alpha + 1) - pref(x)$ of x and the $(|y| - \alpha + 1) - pref(y)$ of y must share at least one token, where α is the minimal overlap between them. Lastly, *position filtering* follows that $Overlap(x, y) \geq \alpha \Rightarrow Overlap(x_l(\omega), y_l(\omega)) + min(|x_r(\omega)|, |x_r(\omega)|) \geq \alpha$, where ω is a particular token that splits records x and y ($\omega \in x \cap y$) into left ($x_l(\omega)$ for x and $y_l(\omega)$ for y) and right partitions ($x_r(\omega)$ for x and $y_r(\omega)$ for y), and tokens in each record are sorted by a particular ordering O. This determines if two records are *similar* based on common tokens in their prefixes and *unseen* right partitions.

Pre-processing: Before applying the filtering procedures, data must be pre-processed: Records are firstly tokenized into token sets using lower-case *bi*-gram tokenization. Additionally, in order to determine if the overlap between two records is adequate when executing PPJoin, tokens and records need to be processed in a certain order. Therefore, tokens from each record are counted (document frequency) and sorted in ascending order (assigning each a position) in a global token ordering O. Utilizing the newly assigned positions in O, tokens in each record are re-ordered accordingly. The last two steps are in preparation for the filtering procedures: for prefix filtering, each record's prefix length is calculated according to $pref(x)$; for length filtering, the records are sorted by their lengths.

Algorithm: The PPJoin algorithm has three phases: *indexing* tokens into an inverted index, *candidate generation* for generating pairs probed on tokens in the prefixes of records from the inverted index, and *verification* for determining if a candidate pair satisfies the overlap constraint. PPJoin constructs the inverted index by inserting tokens from records utilizing their positions. Additionally, a hash map is generated for accumulating overlaps between records. For each record, its prefix is compared against the prefix of another to check if they both contain the same token, and the length filter is applied over them. The satisfied record pair is tested by the prefix and position filters. The overlap stored in the hash map is incremented if the current overlap plus the upper bound (maximum possible overlap of the right partition) is greater than equal to α.

For each record, the hash map contains the number of overlapping tokens between all other records. It then verifies if the candidate pair has enough overlap to be considered a true pair. The heuristic is to use the last token in the prefix of each record as a pivot, where only the record with the smaller token in the suffix needs to be intersected with the other record. If the resulting

intersection count plus the original overlap is still greater than α, the pair is added to the set of true pairs.

2.4. Privacy-Preserving PPJoin: P4Join

P4Join (Privacy-Preserving Prefix Position Join) [30] is a privacy-preserving variant of PPJoin, converting records into fingerprints or same-sized bit arrays.

The basic hashing method h_i for record encryption is defined as: $h_i(x) = (f(x)+i \cdot g(x)) \mod l$, where x is a set of a n-gram tokens generated from a record, f and g are primitive hash functions and l is the length of the fingerprint. Specifically, f is HMAC-SHA1 and g is HMAC-MD5. Each token is hashed into a bit array by applying h_i multiple times where each h_i ($i = 1$ to k) sets one bit in the array. Thus a token is represented by the k set bits, all corresponding to the final fingerprint. This method is applied on all following tokens to complete computing the bit array. Given this new representation, Jaccard similarity can still be applied as a variant known as Tanimoto similarity: $Sim_{Tanimoto}(x, y) = \frac{|x \wedge y|}{|x \vee y|} = \frac{|x \wedge y|}{|x|+|y|-|x \wedge y|}$, where $|x|$ and $|y|$ denotes the number of set bits (cardinality) in the bit array x and y respectively.

Pre-processing: In very similar fashion as PPJoin, P4Join's pre-processing requires tokens to be reordered, prefixes to be calculated and records to be sorted according to length. With Tanimoto similarity, token frequency is calculated based on set bit positions; token reordering is then done according to frequency in ascending order. Prefixes here, additionally, refer to the number of set bits as well, as opposed to the size calculated in $pref(x)$. And lastly, records are sorted by ascending set-bit cardinality.

Algorithm: Algorithmically, P4Join does *not* construct an inverted index because the cost of maintenance outweighs the achievable savings on bit arrays. Instead, it maintains a mapping from set-bit cardinality to all relevant record ids. Then, the three filters are *optionally* applied and pairs that pass them are evaluated via Tanimoto similarity.

2.4.1. Issues. One incurred penalty introduced in P4Join is that of *accuracy*: while PPJoin is a loss-less approach that produces exactly the same result as Jaccard similarity, P4Join is *not*. This is because P4Join leverages *Bloom-filter* like encoding methods that one-way hash the records into *fixed-length* fingerprints no matter the length of the originating record. Furthermore, tuning the parameters k and l with large amounts of data is a *non-trivial* task. Though the original P4Join paper does not suggest how to choose these parameters optimally, real-world implementations using Bloom-filters [23], [25] give us the following false positive rate: $f \approx (1 - e^{-\frac{Uk}{l}})^k$, where U indicates the total number of unique tokens. Thus, inducing the optimal k to be $k_{opt} = \frac{l}{U} ln2$. A lower false positive rate f provides better linkage quality, but in turn provides poorer privacy. So, even if k and l can be adjusted to an optimal combination, hash collisions remain an *unavoidable* issue.

Moreover, we perform thorough experiments regarding parameter tuning and filtering efficiency over our own implementation of P4Join. From the experiments, the improved efficiency provided by the three filters in PPJoin is *not* realized in P4Join, and in many instances applying said filters can be *worse* than running a full comparison of Tanimoto similarity. Specifically, the overhead incurred by the position filter noticeably slows down execution.

Lesson 1: *Adapting string level filtering into bit level directly, or even with modifications is not as efficient, since the bit operations are faster before translation.*

2.4.2. Security Analysis. Though P4Join seeks to introduce privacy protections as they pertain to PPJoin, the effects are *limited*, leaving record data vulnerable.

In particular, P4Join employs one-way hashing via Bloom-filter encodings as its main privacy tool. While such a technique may be an efficient (or first-pass) attempt at privacy-protection, it does not provide any *strong* privacy guarantees and is *provably insecure* in certain scenarios. For example, in cases where the universal set of elements (in this case tokenized *bi*-grams) is *enumerable*, a simple **brute-force** attack is possible where an adversary with sufficient computational resources can simply check each possible element and reconstruct back the filter's content [4]. In such a case, the false-positive rate (f) acts as the *only* (weak) barrier for privacy protection, introducing ambiguity in the enumeration process.

A **simulation** attack is also possible on P4Join. Building off analysis in [14], [32], when dealing with bit-array fingerprints, the sensitivity of each bit can be learned, directly revealing the underlying hashed content. If a bit position represents a large number of tokens, and if these tokens correspond to the same record (i.e. higher sensitivity), the more vulnerable the content of the corresponding record will be [32]. In this way, an adversary can easily guess the content present in the underlying records via an understanding of the sensitivity of certain bit positions.

Furthermore, as mentioned, the parameters k and l need to be selected properly as their relationship is inversely proportional, representing the trade-off between accuracy and privacy as explained by the false positive rate f and optimal k. For example: k and l are both integers, where $k << l$. In this scenario, it is highly possible (if not likely) that hashes for each record represent a unique fingerprint. Given uniqueness, with **misconfiguration**, even passive adversaries who just monitor data transmission channels can easily figure out which bits represent which tokens via simple cross-checking (e.g. eavesdropping). In active adversary (e.g. man-in-the-middle) settings this problem can be exacerbated if the computation unit is hacked or independent connections are formed for relaying impersonated messages.

Lastly, since the fingerprint generation process in P4Join is not aided with any random noise or salt, the same record content generates the exact same fingerprint every time. With enough time for information collection, an adversary could easily map fingerprints back to plaintext via the construction of a **rainbow table** for low-cost future attacks. This in combination with the unprotected communication channels between parties can lead to data sniffing, interception, and manipulation.

Lesson 2: *The hash-based privacy leveraged by P4Join is not as secure as it looks; even compromising on linkage quality provides only weak privacy guarantees.*

2.5. Homomorphic Encryption

2.5.1. The BGV scheme. We work with the BGV [6] (Brakersi, Gentry, Vaikuntanathan) fully homomorphic construction. BGV is an ideal choice for our use case due to its ability to support SIMD (single instruction multiple data) operations over its plaintexts and integer arithmetic

over the learning with errors (LWE) [28] instance, as well as the corresponding ring (RLWE [24]) instance. Homomorphic encryption [2] can be summarized as a quartet of algorithms ($KeyGen$, E, D, f) where $KeyGen$ outputs a key-pair (pk, sk) containing a public key (pk) and a private key (sk), E is an encryption algorithm that takes as input the public key pk, message m, and outputs the ciphertext c, D is a decryption algorithm that takes as input the private key sk, ciphertext c and outputs the message m, and finally, f is an evaluation function evaluated over ciphertexts outputting c_f (an encrypted computation). The *homomorphic* property of HE refers to the upholding of the following: $E(m_1) * E(m_2) = E(m_1 * m_2)$ $\forall m_1, m_2 \in M$, where $*$ represents a homomorphic operation, and M represents the plaintext space.

Security properties: HE schemes, are proven to be *semantically* and IND-CPA [3], [15] secure given the following scenario: provided two equally likely, distinct messages m_1, m_2 and a ciphertext c, no adversary should have an advantage in guessing (with probability $p > \frac{1}{2}$), whether c is an encryption of m_1 or m_2 [7]. Furthermore, with respect to *semi-honest* adversaries [16], the BGV scheme can be extended to provided *evaluation privacy*; this refers to the security and preservation of the operations applied over a ciphertext c, such that an adversary cannot tell what was performed to obtain c.

Lesson 3: *Li and Micciancio in [21] have recently shown that the IND-CPA security model is insufficient for approximation encryption schemes (e.g. CKKS [20]), potentially revealing the secret key. Developing accurate and efficient protocols for 2+ parties encrypting data under a joint key is increasingly difficult, and tradeoffs between accuracy and security need to be carefully considered.*

2.5.2. Threshold HE. While PPJoin isn't inherently a distributed protocol, translating it to be *private* naturally requires secure multi-party computation [34]. Motivating the use of HE in such scenarios can be done with the help of threshold-HE [13], [29].

In threshold-HE, $KeyGen$, E, and D in the quartet of algorithms used to describe HE are replaced by distributed key generation (DKG), distributed encryption (DE), and distributed decryption (DD) protocols, converting key generation and decryption into *interactive* processes. In particular, DKG generates a public (pk), secret (sk) key-pair, where sk is split and distributed into n secret shares, one-per-party. DE uses pk on behalf of all parties, and DD is done collectively by merging the results of each party's individual decryption. The threshold parameter d (where $d \leq n$) determines how many secret shares of sk are needed to correctly perform DD.

Security properties: Threshold homomorphic encryption is also provably *semantically* secure, given the following scenario: provided n parties and a secret key sk, no adversary, given secret key shares $sk_{i \in S}$ for a set S with a dimension $< d$ (where $d \leq n$), should gain any additional advantage in recovering sk [18].

3. HE-PPJoin

3.1. Protocol

Our protocol (Figure 2) involves two forms of pre-processing before *private* versions of `PPJoin` and `Verify` (algorithms 1 and 2 in [33]) can be applied.

Algorithm 1: Helper functions

```
1  Function KeyGen(kp_1 ... kp_n):
2      kp_mp ← MultipartyKeyGen(kp_1 ... kp_n);
3      return kp_mp;
4  Function IsMatch(kp_mp, Enc(t_i), Enc(t_j)):
5      sub ← (Enc(t_i) − Enc(t_j)) · r_*;
6      return (Decrypt(kp_mp, sub) == 0);
7  Function DocFreqJoin(kp_mp, O_1 ... O_n):
8      O ← {}
9      foreach Enc(t_i) ∈ O_m do
10         foreach Enc(t_j) ∈ O_n do
11             if IsMatch(kp_mp, Enc(t_i), Enc(t_j)) then
12                 O[t_j] += 1;
13             else
14                 O[t_i] = 1;
15     return O;
   // PSI: private set intersection
16 Function PSI(kp_mp, x, y):
17     psi ← 0
18     foreach Enc(t_i) ∈ x do
19         foreach Enc(t_j) ∈ y do
20             psi += IsMatch(kp_mp, Enc(t_i), Enc(t_j));
21     return psi
```

Algorithm 2: HE-PPJoin (R, t)

```
   // Local pre-processing
1  kp_mp ← KeyGen(kp_1 ... kp_n);
2  D_1 ... D_n ← Encrypt(kp_mp, t_i for t_i ∈ r_j ∈ D_k);
3  O_1 ... O_n ← Encrypt(kp_mp, t_i ∈ O_j);
   // Global pre-processing
4  R ← {D_1 ... D_n};
5  O ← DocFreqJoin(kp_mp, O_1 ... O_n);
   // PPJoin
6  S ← ∅, I_w ← ∅;
7  foreach x ∈ R do
8      A ← empty map from record id to int;
9      p ← |x| − ⌈t · |x|⌉ + 1;
10     for i = 1 to p do
11         w ← x[i];
12         foreach (y, j) ∈ I_w such that
13         IsMatch(kp_mp, w, y[j]) and x, y ∉ D_k
14         and |y| ≥ t · |x|) do
15             α ← ⌈(t/(1+t))(|x| + |y|)⌉;
16             ubound ← 1 + min(|x| − i, |y| − j);
17             if A[y] + ubound ≥ α then
18                 A[y] ← A[y] + 1;
19             else
20                 A[y] ← 0;
21         I_w ← I_w ∪ {(x, i)};
22     HE-Verify(x, A, α);
```

Algorithm 2 and algorithm 3 extend said algorithms to include local & global pre-processing steps, data structure modifications, as well as other algorithmic adaptations. The explicit differences composed in our private algorithms are highlighted in yellow for clarity.

Local pre-processing: The first step for each party $(P_1 \dots P_n)$, is *key generation*. All parties must first establish consensus on a cryptographic scheme of choice. Once established, parties interact to compute a joint public (evaluation) key that corresponds to the result of all their secret shares $(sk_1 \dots sk_n)$, each share retained by its corresponding party. This process is represented by line 1 in algorithm 2 via the `MultipartyKeyGen` (lines 1-3) procedure presented in algorithm 1; kp_{mp} represents the multiparty key-pair containing the joint evaluation key

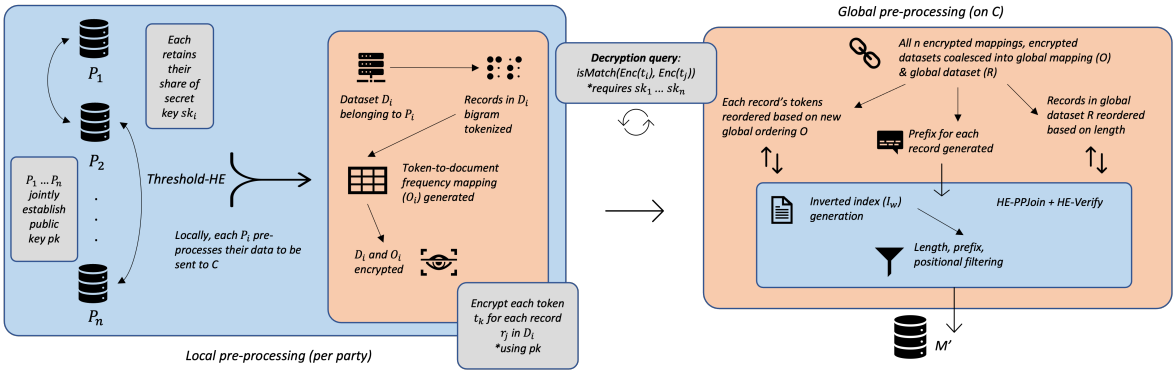

Figure 2. HE-PPJoin. Our protocol is split out into three main components: local pre-processing, global pre-processing, and the `HE-PPJoin` & `HE-Verify` algorithms. Local pre-processing is per-party utilizing threshold-HE with input p_k, whereas the global pre-processing and the algorithms are performed over a computation host utilizing p_k and each party's encrypted data. `IsMatch` queries require a subset of all secret shares.

Algorithm 3: `HE-Verify` (x, A, α)

```
1  foreach y such that A[y] > 0 do
2      Enc(x_l), Enc(y_l) ← last tokens in resp. prefixes;
3      foreach Enc(t) ∈ O.keys do
4          if IsMatch(kp_mp, Enc(t), Enc(x_l)) then
5              w_x = pos. of Enc(t) in O;
6          if IsMatch(kp_mp, Enc(t), Enc(y_l)) then
7              w_y = pos. of Enc(t) in O;
8      overlap ← A[y];
9      if w_x < w_y then
10         ubound ← A[y] + |x| − p_x;
11         if ubound ≥ α then
12             overlap ← overlap + PSI (
13             kp_mp, x[(p_x + 1) ... |x|], y[(A[y] + 1) ... |y|]);
14     else
15         ubound ← A[y] + |y| − p_y;
16         if ubound ≥ α then
17             overlap ← overlap + PSI (
18             kp_mp, x[(A[y] + 1) ... |x|], y[(P_y + 1) ... |y|]);
19     if overlap ≥ α then
20         S ← S ∪ (x, y);
```

pk and the secret shares $sk_1 \dots sk_n$. Once key generation is complete, each P_i completes the following: records in their dataset D_i are *bi*-gram tokenized and a token-to-document frequency map (O_i) is created. When processed, D_i and O_i are ready to be encrypted via pk: each token in each record in D_i is encrypted, and likewise each key (token) in the map O_i is encrypted (lines 2-3 in algorithm 2). Each party then sends these locally encrypted data structures over to a computation host C.

Global pre-processing: Once C receives pairs of encrypted data (D_i, O_i) from each party, they are coalesced into global formats (lines 4-5 in algorithm 2). Coalescing into R (the final sorted record multi-set) is quite simple. however, coalescing all document frequency maps $(O_1 \dots O_n)$ into O is trickier since all keys are now encrypted. We thus employ the `DocFreqJoin` helper method (lines 7-15 of algorithm 1) to securely tabulate the document frequency for all encrypted tokens amongst all maps. This procedure, in turn, utilizes another helper function `IsMatch` (lines 4-6 of algorithm 1) which can be seen as an interactive decryption query that performs a homomorphic subtraction between two tokens, multiplied by a uniformly random non-zero ciphertext element (generated on any of the P_i, and encrypted via pk) in order to determine if the plaintext tokens underlying have the same value. This query performs the distributed decryption (DD) algorithm that requires a subset of secret keys $(s_1 \dots s_k)$. Thus, for two tokens, C receives back either 0 (if they match) or a uniformly random non-zero plaintext value (if they do not match), revealing *no* information about the encrypted tokens *or* their relationship with one another. Once R and O are formed, each record's tokens are individually sorted based on the position assigned to them in O, prefixes for each record are generated, and records in R are ordered by record size.

Note that the `IsMatch` decryption query is the *only* interactive procedure we allow C to perform. The use of the uniformly random ciphertext element randomizes the decryption such that C cannot recover any relationship between t_i, t_j unless they match (query returns 0), in which case the underlying value of the token is still obscured.

Lesson 4: *Allowing interaction in a HE-based scheme is not straightforward, yet it can still be effective if we ensure its usage does not leak any information.*

Algorithm: After pre-processing, our privacy-preserving adaptations of `PPJoin` and `Verify` can be applied over encrypted data seamlessly. In particular, line 14 of algorithm 2 applies `IsMatch` over a token from x's prefix (w) compared with a token from record y $(y[j])$. This operation is necessary to determine whether two encrypted tokens *overlap* between the prefixes of two records in the inverted index I_w. Additionally, we apply an extra filtering step that does not exist in the original `PPJoin` algorithm: we require that x and y *do not* belong to the same dataset D_k. This step is taken for the removal of extra comparisons and to remain consistent with our problem definition as it relates to ER – we seek to find true pairs *across* datasets, not those within the same dataset. While this formalism derives naturally from the concept of different data owners, this is not a concern in PPJoin.

In algorithm 3, the first main difference is in determining w_x, w_y. When extracting out the last tokens in the prefixes of x, y, since they are encrypted, there is no way to directly get their document frequency or position from O. While all the tokens in x, y naturally are a part of the key-set in O, it goes without saying that there is *no* direct equality comparison possible between the

two encrypted values, and so `IsMatch` is once again used to determine if their underlying values correspond to the same plaintext (lines 2-7). Besides this, the only other change necessary is in the calculation of the overlap between right-hand partitions of the x, y (lines 12-13 and lines 17-18). To accomplish this, we apply `PSI` (lines 16-21 of algorithm 1) to determine the number of overlapping elements between two inputted sets of encrypted tokens; the formulas for for the subsets of x and y match exactly to those of the original `Verify` algorithm.

3.2. Optimizations and Efficiency

3.2.1. Multi-threading, RNS, and Batching & SIMD. We harness the following performance benefits of PALISADE and the BGV scheme: First off is multi-threading, which comes in-built to the platform provided by OpenMP [12]. Next, since PALISADE implements BGV in it's *rns* or residue number system variant, computing parallel sets of $<$ 64 bit integers (involving residues) equating to corresponding larger integers is directly optimized for computation on 64-bit architectures. Lastly, batching, also utilized in BGV, is the idea that evaluating a function f homomorphically over l blocks of data can be done approximately as fast as evaluating f on a single encrypted input. This is because ciphertexts can correspond to more than one plaintext slot, allowing for *parallelization*, lowering the per-gate operation cost [31]. This harnesses the power of SIMD (single instruction, multiple data) allowing for elements to be loaded and instructions to be run on CPU registers in parallel.

3.2.2. Cuckoo Hashing. In profiling our different algorithmic components, the naive PSI protocol used by our adaptation in subsection 3.1 as well as `DocFreqJoin` both suffer from heavy incurred overhead. If similar elements could be coarsely clustered with a low-cost algorithm, the overhead would be drastically reduced.

Chen [9] proposed an enhanced variant based on the naive PSI protocol by employing cuckoo hashing. Cuckoo hashing [27] uses a set of simple hash functions $H_1 \dots H_c$ to hash a target x into one of n hash tables with a constant time cost for lookup and deletion and an amortized $O(1)$ cost for insertion. It achieves this by randomly choosing an index $i \in [1, c]$ and inserting x at the location $H_i(x)$. However, insertions may fail due say y already occupying an attempted insert location. In such cases, the algorithm imitates the behavior of a cuckoo bird: the old element y is replaced by x and placed at j, a randomly chosen new index. Since c, the number of hash functions, is constant, lookups and deletions are also constant whereas insertions are amortized $O(1)$ because, with high probability, a vacant position will be discovered for re-insertion. However, if a cycle is detected, all the hashing tables need to be recomputed with a new set H'.

Applying cuckoo hashing directly to PSI, however, requires non-trivial modifications: in Chen's two-party protocol, the receiver executes cuckoo hashing where all elements are hashed into *one* table, whereas the input of the sender is encoded by all available hash functions and the results are stored in a *two*-dimensional table, so that the sender does not need to know exactly what hash function is used by the receiver for each element. In addition, in end-to-end HE adaptations, the use of cuckoo hashing can be quite cumbersome. For example, `DocFreqJoin` in our protocol aggregates matching tokens into one bin and represents them by a single ciphertext as the key for O. This mechanism is more analogous to a simple hashing scheme that would in fact take less execution time.

Lesson 5: *The proposed use of cuckoo hashing can be interpreted as a kind of local sensitive hashing (LSH) where potentially similar items are mapped to the same bin. Integrating this variant requires non-trivial modifications, and running local experiments showed that a more mature LSH instead might be a more comprehensive solution.*

Lesson 6: *The use of HE end-to-end is not always a universal solution; other privacy-preserving methods may better fit the scenario depending on what is important.*

3.3. Correctness

The correctness of our adaptation is built off of BGV [6] and threshold-HE [11], [18]. Local preprocessing (line 1 of algorithm 2) utilizes DKG and DKE. DKG takes in the number of parties (n), and the threshold parameter (d). The corresponding output is a vector of secret keys ($s_1 \dots s_n$), and a public evaluation key pk (each party receiving (pk, sk_i)). DE (lines 2-3 of algorithm 2), is where each P_i encrypts their D_i and their local O_i. The inputs are each party's secret key (sk_i) and a token (t_j), and the output is a ciphertext c_j. DD is used during calls of the `IsMatch` (lines 14 of algorithm 2 and 4 & 7 of algorithm 3) function. Being interactive, requiring a subset of parties ($k < n$), the inputs are ($s_1 \dots s_n$), the threshold parameter (d), and a ciphertext c_j. The corresponding output is a decrypted plaintext element m_j. However, DD is never used to directly decrypt the contents of any encrypted token; rather, we only allow decryption of the randomized encrypted quantity computed with the help of lines 4-6 of algorithm 1.

Lesson 7: *The correctness requirement [2], here relies on the DD: if at least d parties do not perform adversarially, then the resulting outputs are provably correct [5], [18]. Adapting a plaintext algorithm to the private domain with HE requires extensive handling to ensure amongst various attack vectors, the output is obfuscated, but correct when compared against its non-private counterpart.*

3.4. Privacy

While we claim *no* leakage of fatally sensitive information, it is important to formalize this by noting what we do leak given a semi-honest (or *honest-but-curious* [16]) adversary $\mathcal{A}$. We succeed in obfuscating record content as well as their corresponding tokenized representations, document frequency scores, prefixes, and sorted orderings. Additionally, just like in `PPJoin`, we do *not* reveal any information regarding similarity scores of (non-)similar records. Furthermore, since M', the output, includes records that are *similar*, it is not our goal to preserve knowledge of this from C. However, since `IsMatch` is never fully applied directly over the entirety of two token sets (records) r_i, r_j and only applied over encrypted prefixes, C can never uncover the specific similarity score between them. We do, though, leak the size, or number of tokens in each record to C, since the use of encryption is applied at the granularity of tokens. Lastly, we do *not* consider the effects of possible inference attacks by C because of its inability to actively probe with arbitrary inputs and the requirement to not deviate from the protocol given a semi-honest setting.

Lesson 8: *In this work, we aim to achieve a level of provable security to solve ER tasks where data sensitivity is a critical issue. However, this goal is pervasively at odds with performance. It is essential to design, evaluate adversary models, and profile PPER algorithms in a way that makes sense for real-world utility, while maximizing as much privacy preservation as possible.*

4. Experiments

4.1. Experimental Setup

Environment settings: All our experiments were run on a virtual Ubuntu 18.04.4 LTS server, with 2 CPUs from Intel Xeon CPU E5-2690 v4 @ 2.60GHz and 4GB memory.
PALISADE (OpenFHE): We implemented our protocol in PALISADE [1] (version 1.11.5), an open-source, lattice based cryptographic library. The configuration for BGV scheme: *multiplicative depth* = 1, *plaintext modulus* = 65537, *sigma* = 3.2, *security level* = 128 bits. With respect to threshold-HE, our experiments utilize an $n = 2$.
P4Join: To the best of our knowledge, no existing implementation of P4Join is publicly available. Thus, we implemented our own, open sourcing it under MIT license.
Datasets: We utilize *Febrl*, a synthetic dataset generated using *dsgen* [10], to evaluate run-time performance and to assess accuracy. We choose Febrl due to its synthetic nature, meaning that the dataset size and noise introduced are user-controllable. We prepare 2 different-sized sets (100, 500) records (each in 3 different threshold values, t = 0.2, 0.5, 0.8) for evaluation with the following splits for two data owners D_1, D_2: (20/80), (100/400). We report precision, recall and f-score, which are commonly used for evaluating performance in entity resolution tasks.

4.2. Results

We perform time-cost benchmarking for each of the three main components of HE-PPJoin on two prepared Febrl datasets (Febrl-100, Febrl-500) utilizing three different threshold levels ($t = 0.2, 0.5, 0.8$). The results along with a comparison against P4Join are shown in Figure 3 and Figure 4. For HE-PPJoin, we compare against a baseline full comparison, which refers to a HE Jaccard-based private set intersection algorithm described in [35].

For Febrl-100, when $t = 0.2$, the overall running time is around 30 minutes, whereas when $t = 0.5$, the running time drops to $\sim$ 15 minutes, and when $t = 0.8$, it drops even further to $<$ 10 minutes. HE-Jaccard proves to be by far the most costly, taking over an hour to run all 1,600 ($20 \cdot 80$) comparisons. For Febrl-500, the same trend follows: when $t = 0.2$, execution takes $>$ 600 minutes, dropping to <400 minutes, and $\sim$ 150 minutes for $t = 0.5$ and $t = 0.8$, respectively. HE-Jaccard for Febrl-500 again proves to be extremely time-consuming, >1,600 minutes for all 40,000 ($100 \cdot 400$) comparisons. In terms of individual components, local pre-processing cost is negligible (in scale of $\sim$ 30 seconds) as compared with the time cost of global pre-processing and the time for HE-PPJoin and HE-Verify. Furthermore, the results show that overall, global pre-processing time remains constant regardless of the threshold value t, and purely depends on the dataset size as shown. The time cost for HE-PPJoin & HE-Verify, however, is strictly dependent on t and can drastically improve (closer to $t = 0.8$) or worsen (closer to $t = 0.2$) given certain choices for it.

Compared with P4Join side-by-side, the added overhead of HE-PPJoin, incurred due to the private HE operators, is clearly *significant*. However, keeping context and baselines in mind, it is important to note that comparatively P4Join does *not* achieve expected time cost savings as compared with its *own* baseline (Tanimoto).
Lesson 9: *While the added cost of HE is non-trivial, irrespective of t's value, HE-PPJoin still shows significant time cost savings as compared to its baseline (HE-Jaccard), meaning the algorithmic benefits remain in-tact even in the private domain with a proper HE adaptation.*

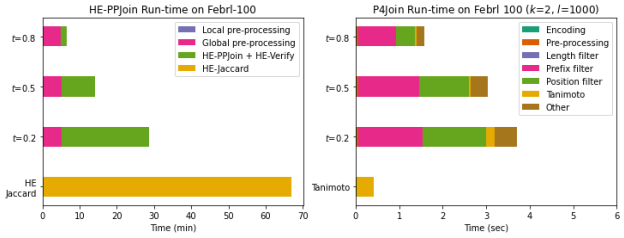

Figure 3. HE-PPJoin vs P4Join run-time on Febrl-100

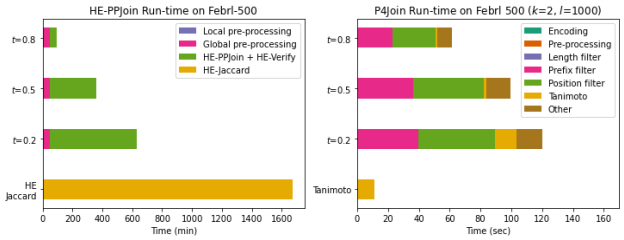

Figure 4. HE-PPJoin vs P4Join run-time on Febrl-500

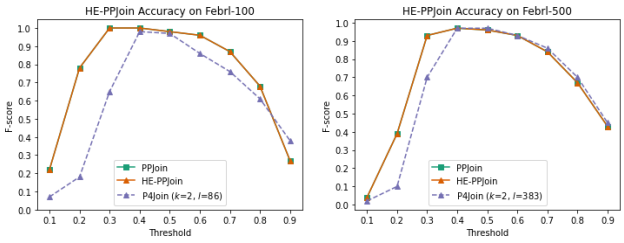

Figure 5. HE-PPJoin vs P4Join accuracy

4.2.1. Accuracy analysis. We additionally analyze the accuracy of our adaptation as compared with P4Join over both Febrl-100 and Febrl-500. As shown in Figure 5, HE-PPJoin's F-score matches that of PPJoin's for *all* threshold t ($0.1 \leq t \leq 0.9$) values. This is due to the fact that we apply the usage of the three filtering techniques: length, prefix, and position in the *exact same* algorithmic format as that of the original PPJoin and Verify algorithms. Furthermore, since BGV works with exact integer arithmetic, the results of DD are ensured to be exact matches with their corresponding plaintext values (i.e. `IsMatch` incurs no approximation error). On the other hand, P4Join (with parameters of $k = 2$ and $l = 86$), we can clearly see an accuracy drop-off in Febrl-100 (for $0.1 \leq t \leq 0.4$ and $0.6 \leq t \leq 0.8$) and in Febrl-500 (for $0.1 \leq t \leq 0.4$).
Lesson 10: *The proper HE adaptation is able to not only maintain the performance benefits, but also does not compromise the accuracy of the output, regardless of the parameters.*

5. Conclusion

In this work, we present HE-PPJoin, a privacy-preserving adaptation of PPJoin using the BGV FHE construction. We describe the run-time performance and accuracy of our approach, while also assessing the correctness and privacy of our adaptation in the presence of a semi-honest adversary. Furthermore, we present a similar, rigorous analysis of these same properties as they relate to P4Join, an existing variant of PPJoin. Although our protocol introduces a non-trivial amount of overhead, we still see *practical* performance execution for small dataset sizes, while ensuring *no* information leakage.

While the use of HE for work in applications such as PPER is still relatively new and inefficient, many recent works have established directions to bridge this gap for usability in the real world [17], [22]. The use of such new directions with our existing protocol can definitely be explored to scale to more complex, real-world data.

References

[1] PALISADE Lattice Cryptography Library (release 1.11.5). https://palisade-crypto.org/, 2021.

[2] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption standard. Cryptology ePrint Archive, Report 2019/939, 2019. https://eprint.iacr.org/2019/939.

[3] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 394–403, 1997.

[4] Giuseppe Bianchi, Lorenzo Bracciale, and Pierpaolo Loreti. "better than nothing" privacy with bloom filters: To what extent? pages 348–363, 09 2012.

[5] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 565–596, Cham, 2018. Springer International Publishing.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. https://eprint.iacr.org/2011/277.

[7] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Security of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, USA, July 2017.

[8] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 5–5. IEEE, 2006.

[9] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1243–1255, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Peter Christen. Probabilistic data generation for deduplication and data linkage. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 109–116. Springer, 2005.

[11] Ronald Cramer, Ivan Damgård, and Jesper B. Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, pages 280–300, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[12] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[13] Yvo Desmedt. *Threshold Cryptography*, pages 1288–1293. Springer US, Boston, MA, 2011.

[14] Elizabeth A Durham, Murat Kantarcioglu, Yuan Xue, Csaba Toth, Mehmet Kuzu, and Bradley Malin. Composite bloom filters for secure record linkage. *IEEE transactions on knowledge and data engineering*, 26(12):2956–2968, 2013.

[15] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[16] Carmit Hazay and Yehuda Lindell. Semi-honest adversaries. 2010.

[17] Ilia Iliashenko and Vincent Zucca. Faster homomorphic comparison operations for bgv and bfv. Cryptology ePrint Archive, Report 2021/315, 2021. https://ia.cr/2021/315.

[18] Aayush Jain, Peter M. R. Rasmussen, and Amit Sahai. Threshold fully homomorphic encryption. Cryptology ePrint Archive, Report 2017/257, 2017. https://eprint.iacr.org/2017/257.

[19] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String similarity joins: An experimental evaluation. *Proc. VLDB Endow.*, 7(8):625–636, apr 2014.

[20] Yongsoo Song Jung Hee Cheon, Andrey Kim & Miran Kim. Homomorphic encryption for arithmetic of approximate numbers. 2017.

[21] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. Cryptology ePrint Archive, Report 2020/1533, 2020. https://ia.cr/2020/1533.

[22] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping. Cryptology ePrint Archive, Report 2021/1337, 2021. https://ia.cr/2021/1337.

[23] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials*, 21(2):1912–1949, 2018.

[24] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Report 2012/230, 2012. https://eprint.iacr.org/2012/230.

[25] DZEJLA MEDJEDOVIC. *Algorithms and data structures for massive datasets*. O'REILLY MEDIA, 2022.

[26] Matthew Michelson and Craig A. Knoblock. Learning blocking schemes for record linkage. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 440–445. AAAI Press, 2006.

[27] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, *Algorithms — ESA 2001*, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[28] Oded Regev. The learning with errors problem (invited survey). In *2010 IEEE 25th Annual Conference on Computational Complexity*, pages 191–204, 2010.

[29] Berry Schoenmakers. *Threshold Homomorphic Cryptosystems*, pages 1293–1294. Springer US, Boston, MA, 2011.

[30] Ziad Sehili, Lars Kolb, Christian Borgs, Rainer Schnell, and Erhard Rahm. Privacy preserving record linkage with ppjoin. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, 2015.

[31] Nigel Smart and Frederik Vercauteren. Fully homomorphic simd operations. *IACR Cryptology ePrint Archive*, 2011:133, 01 2011.

[32] Dinusha Vatsalan and Peter Christen. Multi-party privacy-preserving record linkage using bloom filters. *arXiv preprint arXiv:1612.08835*, 2016.

[33] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):1–41, 2011.

[34] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[35] Yixiang Yao, Tanmay Ghai, Srivatsan Ravi, and Pedro Szekely. *AMPPERE: A Universal Abstract Machine for Privacy-Preserving Entity Resolution Evaluation*, page 2394–2403. Association for Computing Machinery, New York, NY, USA, 2021.