RC25544 (IRE1506-002) June 3, 2015
Computer Science

IBM Research Report

FPVI: An Efficient Method for Discovering Privacy Vulnerabilities in Datasets

Aris Gkoulalas-Divanis, Stefano Braghin
IBM Research
Smarter Cities Technology Centre
Mulhuddart
Dublin 15, Ireland

FPVI: An Efficient Method for Discovering Privacy Vulnerabilities in Datasets

Abstract—**Analyzing datasets to discover privacy vulnerabilities is an important step in the privacy-preserving data publishing process and an area of increased interest for commercial data masking products. In this paper we propose FPVI, a fast algorithm for discovering privacy vulnerabilities in datasets in the form of combinations of attributes' values leading to few records. FPVI operates in a multi-threaded fashion to efficiently index the data and scan different attributes' combinations in parallel, while pruning the search space to limit the (exponential) number of attributes' combinations that need to be searched for uniques. Our algorithm fully utilizes the execution environment, supporting hardware configurations spanning from commodity machines to multi-CPU multi-core nodes in cluster environments. Through experimental evaluation, using a large number of real-world datasets, FPVI is shown to significantly outperform the state-of-the-art to the extent that we had to design multi-threaded versions of the state-of-the-art method to form the baseline for our experiments. Performance measurements on the scalability of FPVI indicate that our method can analyze microdata consisting of 11 millions of records and 20 attributes in less than 9 minutes.**

I. Introduction

The automatic analysis of datasets to uncover privacy vulnerabilities is an area that has gained increased attention. Depending on the type of the dataset that needs to be protected and the information that it records about individuals, different types of privacy risks need to be considered and different types of vulnerabilities are sought [1], [2]. The identification of privacy vulnerabilities is an important first step for privacy-preserving data publishing, as it provides the necessary input to syntactic anonymization algorithms [3], to allow for sufficient protection of the individuals' privacy. It is also a very useful tool to validate the conformance of a dataset to a given privacy policy and/or to data protection legislation.

In this paper, we consider relational tables that contain information at the level of individual respondents (Table I). Such *microdata* sets are vulnerable to re-identification attacks, in which adversaries associate records to individuals' identities by linking (through triangulation) the data with external, potentially publicly available, datasets. Such triangulation attacks exploit the uniqueness (or rarity) of certain records in the dataset based on a selected, usually small, number of attributes (called *quasi-identifiers* [3]), and have been proven to be very successful even when all direct identifiers (e.g., social security numbers, national-IDs, etc.), have been removed from the data prior to a data release [3], [4], [5].

One way to protect microdata would be to use differential privacy [6], thereby releasing noisy summary statistics of the data or histograms [7]. This, however, is problematic in cases when data recipients want to explicitly study the anonymized datasets at a record-level and place emphasis on the truthfulness of the reported data values. In such cases, syntactic approaches, operating under the k–anonymity principle [3], are much preferred. These approaches, however, require the specification of quasi-identifiers by the data publisher, a task that is difficult to be performed by non-experts.

In this paper we propose FPVI, a multi-threaded algorithm which aims to automate the privacy-preserving data publishing process by automatically discovering the quasi-identifiers in datasets. To do so, the algorithm computes the minimal combinations of attributes that leading to unique individuals in microdata sets. FPVI supports hardware configurations spanning from commodity machines to multi-CPU multi-core nodes in cluster environments. This makes it ideal for commercial use in data masking products, to meet the data anonymization needs of customers with access to diverse hardware infrastructures. Through extensive experimental evaluation on several real-world datasets, we show that our algorithm significantly outperforms the state-of-the-art, as it can analyze datasets consisting of several millions of records and tens of attributes in a matter of only a few minutes.

The remainder of this paper is organized as follows. Section II presents the related work. In Section III, we provide the background that is necessary for explaining our method and derive the problem statement. Section IV introduces our algorithm for the discovery of privacy vulnerabilities in datasets. Section V contains the experimental evaluation of the proposed algorithm, and Section VI concludes this work.

II. Related Work

The identification of privacy vulnerabilities in the form of sample uniques in microdata (also known as *the uniques problem* [8], [9]) has received significant attention from the statistics and the computer science communities. The statistics community tackled the problem by proposing mathematical

	Birth	**G**ender	**Z**IP	**M**arital status
0	09/64	Female	94139	Divorced
1	09/64	Female	94138	Divorced
2	04/64	Male	94138	Widow
3	04/64	Male	94139	Married
4	03/63	Male	94138	Married
5	03/63	Male	94138	Married
6	09/64	Female	94141	Married
7	09/64	Female	94141	Married
8	05/61	Male	94138	Single
9	05/61	Male	94138	Single

TABLE I: An example dataset

models, such as models borrowed from the theory of population genetics [10], to assess disclosure risk. The goal of this research was to capture the proportion of sample uniques in the data which are also population uniques, hence they can lead to the re-identification of individuals.

More recently, a few computer science approaches were developed to automate the discovery of such vulnerabilities. Unlike statistical approaches to the uniques problem, the computer science methods aim to identify all unique (or rare) records in a dataset, and use this information as an indicator of disclosure risk. Their model of attack is more powerful, as they assume attackers who may know that an individual is participating to a dataset and wish to identify their record.

Takemura in 2002 published the first approach [11] in computer science for identifying uniques. The author proposed a method for identifying the minimum sets of variables in a microdata set, with which a record becomes a sample unique. In addition to uniqueness, the author examined the case of rare records, where a record is identical to few other records with respect to a set of variables. Accordingly, he proposed an algorithm for discovering such unsafe records in microdata sets. The proposed algorithm is memory-demanding and can analyze only small datasets, as it suffers from poor scalability.

In 2005, Elliot et. al [12] proposed SUDA, an algorithm for detecting minimal sample uniques in datasets. SUDA considers all combinations of attribute-value pairs in a dataset, starting from a single attribute and moving level-wise to larger sets of attributes, to identify unique records. When a record is found to be unique for a set of attributes, the record is not considered in any superset of the same attributes.

Similarly to [11], SUDA was shown to suffer from poor scalability [13]. This led Manning et al. to propose SUDA2 [13], which improved SUDA by applying an effective pruning strategy. As experimentally verified by the authors, SUDA2 is orders of magnitude faster than SUDA. The algorithm employs a recursive depth-first search (DFS) strategy to generate combinations of attributes to search for uniques.

The nature of the SUDA2 algorithm allows work to be divided into non-overlapping tasks that can execute in parallel. Accordingly, a few parallel implementations of SUDA2 have been proposed. PSUDA2 [14] operates in a cluster environment and requires the dataset to be replicated to all processing nodes, as each node operates in its local memory in a message-passing paradigm. The load balancing strategy of PSUDA2 is poor, leading to the generation of tasks of unpredictable size and complexity that are assigned to cluster nodes. Var-PSUDA2 [14] improves load-balancing by reducing the variability of the generated task sizes, but the computational cost remains high due to the need of replicating the dataset, the costly message-passing interface for distributing the work among the different processors, and the DFS nature of the algorithm. In [15], Haglin et al. evaluate different parallel implementations of SUDA2 that are designed specifically for an SMP cluster, a Cray MTA2 machine, and a heterogeneous group of workstations connected by LAN. Through experiments they demonstrate that their algorithms outperform Var-PSUDA2, but the work generation strategy remains the major bottleneck. An additional shortcoming of these methods is their requirement for specialized, non-commodity, hardware. This is in contrast to the method that we propose in this paper, which supports a wide range of hardware configurations.

A set of approximation algorithms for finding quasi-identifiers are proposed in [16]. The definition of quasi-identifiers in this work differs from the original definition [3], as the methods aim to find α–distinct and α–separating quasi-identifiers, with low space and time complexity. The developed methods use sampling to reduce complexity, and may fail to discover all the quasi-identifiers of the dataset.

Last, we note that the problem of identifying uniques can be considered as a special case of mining infrequent itemsets [17], [18]. The algorithms that have been proposed in this area assume a set-valued data representation and apply frequency-based pruning criteria (similar to [19]) to effectively discover rare itemsets. The problem, however, of finding minimal sample uniques is more specific than that of finding infrequent itemsets, hence the former methods are more effective on this task, leading to a better performance.

III. Background

Let $\mathcal{D}$ denote a microdata set in the form of a table consisting of $\mathcal{R}$ rows and $\mathcal{A}$ columns (see Table I). Each row (or *record*) $r \in \mathcal{R}$ of dataset $\mathcal{D}$ corresponds to an individual, for whom information is recorded along each column (or *attribute*) $a \in \mathcal{A}$. From now on, we will use notation $\|\mathcal{R}\|$ and $\|\mathcal{A}\|$ to refer to the number of records and the number of attributes, respectively, in dataset $\mathcal{D}$. We will further use notation $\wp(A)$ to refer to the powerset of set A.

Given an attribute $a \in A$, we define by *dom*(a) the domain of values for a, which contains all distinct values of the records $\mathcal{R}$ for this attribute. For each value $u \in$ *dom*(a), we define the *frequency* of u as the number of times that value u appears in attribute a for the records $\mathcal{R}$ of dataset $\mathcal{D}$.

The definition of frequency can be extended to the case of attribute-value combinations, as follows: Given a nonempty subset (or *combination*) of attributes $A = \{a_1, a_2, \ldots, a_n\} \in \mathcal{A}$ and a record r with respective values $\{u_1, u_2, \ldots, u_n\}$ for these attributes, we define the *frequency* of the attribute-value combination for record r and attributes in A (i.e., $a_1 = u_1, a_2 = u_2, \ldots, a_n = u_n$), as the number of records in $\mathcal{R}$ for which this attribute-value assignment holds.

Definition (**j–isolation** [11]) Given a record $r \in \mathcal{R}$ and a nonempty set of attributes $A \in \mathcal{A}$, r is *j-isolated* in A if the frequency of the attribute-value combination for record r and attributes in A is exactly j.

Observation The re-identification risk of an individual in a dataset $\mathcal{D}$ is directly related to the *uniqueness* of the individual in $\mathcal{D}$. Accordingly, individuals who are 1-isolated for a set of attributes in $\mathcal{A}$ are at maximum risk of being re-identified, while j-isolated individuals have a $1/j$ probability of being re-identified. An additional consideration regards the amount of knowledge that attackers need to have to re-identify an individual. On this end, the minimal sets of attributes that

```
Input: (i) An ordered set of attributes $\mathcal{A}$, and (ii) reference to a set of sets of
       attributes containing uniques $\mathcal{B}$
Output: The next combination of generated attributes $A$, or False if there are no
        more attribute sets to generate
1  begin
2      Initialize $\mathcal{S}$ (static) as an empty list;        // Performed only once
3      while $\|\mathcal{S}\| < \|\mathcal{A}\|$ do
           // Increment state $\mathcal{S}$
4          idx = 0;
5          while $\|\mathcal{S}\| > idx$ do
6              $m$ = $\|\mathcal{A}\|$−idx−1;        // Maximum value for
                                                      $\mathcal{S}$[idx]
7              if $m > \mathcal{S}[idx]$ then
8                  $\mathcal{S}$[idx]++;
9                  for i = idx; i > 0; —i do
10                     $\mathcal{S}$[i − 1] = $\mathcal{S}$[i] + 1;
11                 break;
12             else
13                 ++idx;
14         if $\|\mathcal{S}\| = idx$ then
15             $\mathcal{S}[\|\mathcal{S}\|]$ = 0;
16             for i = $\|\mathcal{S}\|$ − 1; i > 0; —i do
17                 $\mathcal{S}$[i − 1] = $\mathcal{S}$[i] + 1;
18         $A = \varnothing$;
19         foreach $s \in \mathcal{S}$ do
20             $A = A \cup \{\mathcal{A}[s]\}$;    // $a$ is the representation
                                                 of $S$ as a set of elements
                                                 of $\mathcal{A}$
           // verify that the state is not banned
21         i = 0;
22         while $i < \|\mathcal{B}\|$ do
23             if $\mathcal{B}[i] = (\mathcal{B}[i] \cap A)$ then
24                 break;
25         if $i \neq \|\mathcal{B}\|$ then
26             return $A$;    // return the set of elements of $\mathcal{A}$
27     return False;
```

Algorithm 1: GenerateAttributeSet

lead to 1-isolated records are of particular interest. Such sets of attributes are largely known as *quasi-identifiers* and are important for syntactic anonymization approaches [4].

In the remainder of this work, we propose a method for solving the following problem:

Problem statement *Given a microdata set $\mathcal{D}$ consisting of $\mathcal{R}$ records and $\mathcal{A}$ attributes, where each row corresponds to a distinct individual, identify all minimal sets of attributes in $\mathcal{A}$ that lead to at least one 1-isolated record.*

IV. The FPVI Algorithm

In this section, we introduce a Fast Algorithm for Privacy Vulnerabilities' Identification (FPVI). FPVI is a multithreaded algorithm for the discovery of quasi-identifiers in large datasets. It employs an approach for generating attribute sets, in an incremental fashion, to search for uniques (presented in Section IV-A) and applies an indexing mechanism to create an index-based representation of the input dataset on which the algorithm operates (discussed in Section IV-B). The main operation of the FPVI algorithm is presented in Section IV-C.

A. Generation of attributes' combinations

Algorithm 1 iteratively generates elements of the lattice representing the power set $\wp(\mathcal{A})$ of the attribute set $\mathcal{A}$. The elements of $\wp(\mathcal{A})$ are generated following a pre-defined (e.g., lexicographic) order and in monotonically increasing order of length, starting with single attributes and ending with the element corresponding to the entire set of attributes. Figure 1 shows the generation process for the dataset of Table I.

An important observation about the operation of the algorithm is that the generation process *should not generate all the elements of* $\wp(\mathcal{A})$ *upfront*, because the number of elements is exponential to the cardinality of $\mathcal{A}$ and certain attribute combinations may not be necessary to examine. To achieve that, the algorithm maintains a static state $\mathcal{S}$ that is incremented every time that it is invoked.

More specifically, the algorithm operates as follows. It takes as input two arguments: (i) a (lexicographically) ordered set of attributes $\mathcal{A}$, and (ii) a set of sets of attributes $\mathcal{B}$. The latter set contains the `banned` attribute combinations, i.e., the set of attributes that have been found to contain uniques. This input is needed to avoid generating supersets of these combinations, as they will also contain uniques.

In line 2, the algorithm initializes the static state $\mathcal{S}$ as an empty list, an operation that is performed only once. In line 3, the size of the list representing the state is tested against the size of the set of attributes. If the state list is as large as the set of attributes, the algorithm returns *False*, notifying the caller of the function that there are no more sets of attributes to generate (line 27). Otherwise, the algorithm increments the state $\mathcal{S}$ as follows: It initializes a pointer to elements of $\mathcal{S}$ to 0, which is the first element of the state. As long as the size of $\mathcal{S}$, which is the number of elements in the list, is greater than idx (line 5), it computes m, that is the maximum value admissible in position idx (line 6). Subsequently, the algorithm tests that the current value in position idx is not greater than m (line 7). If this is the case, the algorithm increments the value in position idx (line 8), sets each element of the list with index less than idx to one plus the value of the element preceding it (lines 9–10), and breaks out of the loop of line 5. Otherwise, it increases idx (line 13). The purpose of this operation is to try to increment always the leftmost entry of $\mathcal{S}$ and to maintain the property that each element is smaller than the one immediately on its right. This guarantees that the sets of attributes are generated based on the selected order.

Line 14 checks the reason for which the flow of execution exited the `while` loop of line 5. If the flow of execution reached line 11, then the body of the `if` of line 5 is skipped. Otherwise, it means that the algorithm generated all the elements of $\wp(\mathcal{A})$ of size $\|\mathcal{S}\|$. Therefore, we need to

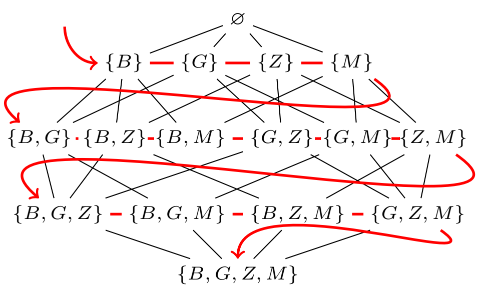


Fig. 1: A lattice and the order of subsets generation

increment the size of $\mathcal{S}$, initializing $\mathcal{S}[\|\mathcal{S}\|]$ to 0, which is the smaller index in $\mathcal{A}$. Subsequently, the algorithm sets the value of all the positions smaller than idx to 1 plus the value to their right. Note that this iteration is done in reverse order, i.e., from (idx − 1) to 0, in order to preserve the updated values. After that (lines 18–20) the algorithm converts the state represented in $\mathcal{S}$ to an actual subset of the attributes of $\mathcal{A}$.

In lines 21–24 the algorithm verifies that the generated subset is not a superset of a set of attributes identified as containing uniques, which are stored in $\mathcal{B}$. If A is not a superset of (and element of) $\mathcal{B}$, then A is returned to the caller. Otherwise, the algorithm discards A and proceeds with incrementing the state $\mathcal{S}$ again (line 3).

Complexity analysis Assuming that the attribute set to be generated has not been banned, the worst-case complexity is when the algorithm computes the combination containing all attributes of $\mathcal{A}$. In this case, the algorithm iterates $(\|A\| - 1)$ times (lines 5–13), with a cost of $O(\|A\|)$ plus $O(\|\mathcal{S}\|)$ (lines 14–17), which in the worst case is $O(\|A\|)$. Note that if the execution flow enters in the `then` branch of the `if` of line 7, it means that the algorithm iterated at most $O(\|A\|)$ times and then iterated again $O(\|A\|)$ times, because of the loop of line 9. Again, this leads to a complexity of $O(\|A\|)$. On the other hand, if $\mathcal{B} \neq \varnothing$ the algorithm also executes the loop of line 22, which has a complexity of $O(\|\mathcal{B}\|)$. Theoretically, $\|\mathcal{B}\|$ can be as large as $2^{\|\mathcal{A}\|}$, but because of how $\mathcal{B}$ is populated $\|\mathcal{B}\| << 2^{\|\mathcal{A}\|}$. More precisely, $\|\mathcal{B}\| \approx \|\mathcal{A}\|$ in the average case. Therefore, the complexity of Algorithm 1 is $O(max\{\|\mathcal{A}\|, \|\mathcal{B}\|\}) \approx O(\|\mathcal{A}\|)$.

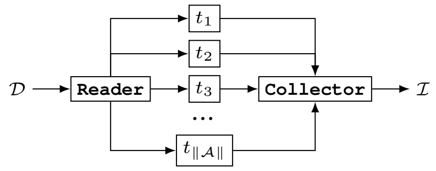


Fig. 2: Create index: process architecture

B. Index creation

In this section, we explain the parallel indexing mechanism that is used by FPVI in order to convert the dataset to a more efficient representation for the discovery of uniques. The indexing process is also illustrated in Figure 2.

```
Input: Dataset $\mathcal{D}$ consisting of $\mathcal{R}$ records and $\mathcal{A}$ attributes
Output: The reverse index $\mathcal{I}$ constructed from $\mathcal{D}$
1  begin
2      $\mathcal{I}$ = [ ] ;                         // Initialise an empty list
3      foreach $a \in A$ do
4          Spawn the worker thread $w_a$;
5      foreach $r \in R$ do
6          $ir$ = [ ];            // Pointers to buckets for record $r$
7          foreach $a \in A$ do
8              Submit($r_a$, $w_a$);
9          foreach $a \in A$ do
10             $ir_a$ = Read($w_a$);
11             Insert($ir_a$, $a$, $ir$);   // Append $ir_a$ to $ir$ in pos $a$
12         Append($ir$, $\mathcal{I}$);
13     return $\mathcal{I}$;
```

Algorithm 2: Create index, main thread

```
Input: $r_a$: value for attribute $a$ on record $r$
Output: $rid$: reference to the bucket in which $r_a$ has been indexed
Data: IDX$_a$: the static index structure for attribute $a$
1  begin
2      if $r_a \notin IDX_a$ then
3          $b$ = Create();           // create a new bucket (set)
4          Index($r_a$, $b$, $IDX_a$);
5      $rab$ = Get($IDX_a$, $r_a$);        // retrieve the bucket for $r_a$
6      Insert($r$, $rab$)
7      return &$rab$;                 // return a reference to $rab$
```

Algorithm 3: Create index, worker thread

Algorithm 2 shows the main thread of the indexing approach. The duty of the main thread is to read the input dataset, split its records in such a way that each worker thread operates only on the part of data it is in charge of, and build a reverse index that links the attributes' values of each record with those of other records in which the same values appear.

Specifically, the algorithm begins by initializing the index (line 2) and then spawns a number of worker threads that is equal to the number of attributes of the input dataset (lines 3–4). The algorithm subsequently reads the dataset line-by-line (lines 5–12), submits each attribute-value to a different worker thread (line 8), and retrieves from the corresponding worker the id of the bucket in which the value for the attribute has been indexed (line 10). Each worker thread (Algorithm 3) is in charge of creating and managing a separate index for an attribute. This way, the costs associated with the indexing process for the attributes of a record, are split among the different worker threads. Note that the main thread retrieves the result of the worker thread for each record in the order of termination, i.e., the main thread does not wait for a specific thread to return a result but it waits for a thread to complete. This way, the population of the list ir is performed efficiently.

Complexity analysis The indexing algorithm executes in parallel a number of operations that are linear to the number of records and to the number of attributes of the dataset. Given that the number of threads t is equal to the number of attributes of the dataset, the operations that need to be performed are $O(\|\mathcal{R}\| \times \|\mathcal{A}\|/t) = O(\|\mathcal{R}\|)$. The cost of each operation depends on the data structure that is used for the indexing. In our implementation we use a tree-based data structure and, thus, the cost of indexing an attribute value is $O(log\|\mathcal{R}\|)$, leading to an overall complexity of $O(\|\mathcal{R}\| \times log\,\|\mathcal{R}\|)$.

Running example Consider the dataset shown in Table I. The algorithm spawns four worker threads. The main thread reads the first record, which has the values `09/64`, `Female`, `94139`, and `Divorced`. Each value is sent to the corresponding worker thread and the index structures are updated as shown in Figure 3a. Specifically, one index per attribute is constructed, currently containing only one value, and the set of records in which that value appears, i.e. $\{0\}$.

Next, the main thread reads the subsequent records and continues to send the various values to the appropriate worker threads. Then, the index evolves as shown in Figure 3b. In this figure, we see the status of the indexing after reading the first four lines of the dataset. One may notice that there

	B	G	Z	M
0	$\{0,1,6,7\}$	$\{0,1,6,7\}$	$\{0,3\}$	$\{0,1\}$
1	$\{0,1,6,7\}$	$\{0,1,6,7\}$	$\{1,2,4,5,8,9\}$	$\{0,1\}$
2	$\{2,3\}$	$\{2,3,4,5,8,9\}$	$\{1,2,4,5,8,9\}$	$\{2\}$
3	$\{2,3\}$	$\{2,3,4,5,8,9\}$	$\{0,3\}$	$\{3,4,5,6,7\}$
4	$\{4,5\}$	$\{2,3,4,5,8,9\}$	$\{1,2,4,5,8,9\}$	$\{3,4,5,6,7\}$
5	$\{4,5\}$	$\{2,3,4,5,8,9\}$	$\{1,2,4,5,8,9\}$	$\{3,4,5,6,7\}$
6	$\{0,1,6,7\}$	$\{0,1,6,7\}$	$\{7,8\}$	$\{3,4,5,6,7\}$
7	$\{0,1,6,7\}$	$\{0,1,6,7\}$	$\{7,8\}$	$\{3,4,5,6,7\}$
8	$\{8,9\}$	$\{2,3,4,5,8,9\}$	$\{1,2,4,5,8,9\}$	$\{8,9\}$
9	$\{8,9\}$	$\{2,3,4,5,8,9\}$	$\{1,2,4,5,8,9\}$	$\{8,9\}$

TABLE II: Record index computed for the dataset of Table I

are three buckets for the attribute M, while only two appear for the other attributes. This because there were, so far, only two distinct values for the first three attributes (B, G and Z), while three appeared in M. Figures 3c and 3d present subsequent evolutions of the indexes. Upon completing the read of the dataset, the main thread returns the reverse index, which contains references to the various buckets. At this point, the original index structures can be deleted to gain memory.

Algorithm 2 does not return the indexes generated for each attribute but the references to the buckets of the indexes, as shown in Table II. Note that Table II is just a simplified representation that we use to explain the algorithm and not the way this information is stored in the index. Namely, each distinct value appearing in a column of Table II corresponds to one memory instance. Thus, there are no duplicates, allowing for an efficient use of the main memory.

C. Main algorithm

In this section, we describe the FPVI algorithm that identifies the set of minimal attributes' combinations that contain uniques in a dataset. FPVI is presented in Algorithm 4 (main thread) and Algorithm 5 (worker thread). It takes advantage of (i) a pruning technique to reduce the search space that the algorithm has to explore, (ii) an approach for generating attribute sets in an incremental fashion (presented in Section IV-A), and (iii) an indexing mechanism to further speed up the analysis of the dataset (presented in Section IV-B).

Algorithm 4 takes as an input a dataset $\mathcal{D}$, the number of threads t and the size BS of each task in number of rows to be searched for uniques. As a first step, the algorithm creates an index $\mathcal{I}$ of the dataset, which stores the unique values appearing in each attribute $\mathcal{A}$ (see Algorithm 2).

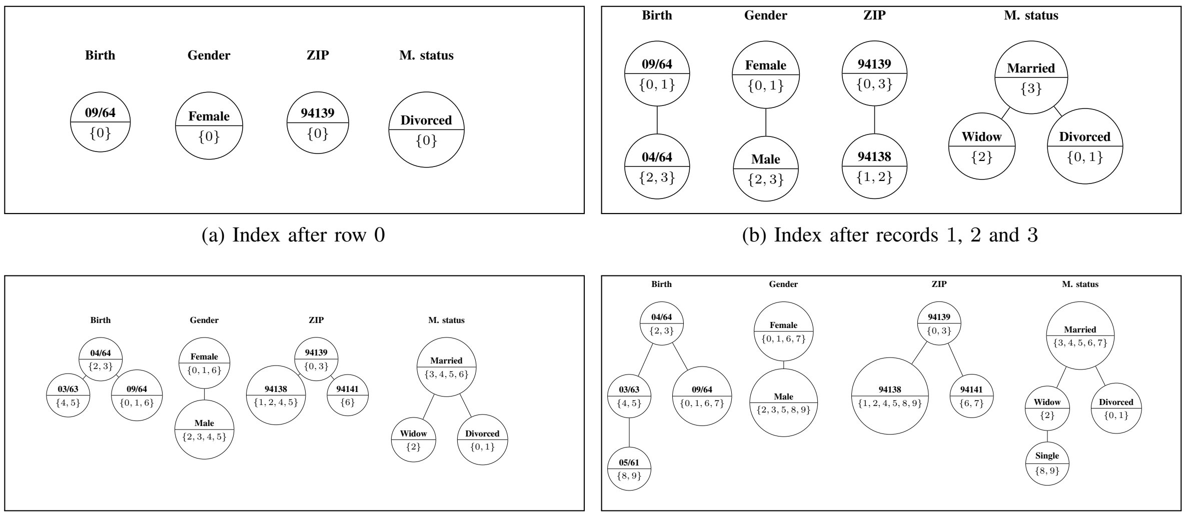
Next, in lines 3 and 4, the algorithm creates t worker threads that have access to two queues, a *task queue* TQ and a *result queue* RQ, as well as index $\mathcal{I}$, offering a compressed version of the dataset. Each created thread will immediately start executing the code described in Algorithm 5.

In line 5, we instantiate AT: a list of pairs (as, rb), where as is a combination of attributes and rb is the last analysed block size for as. In the loop shown in lines 6–9, we generate $t * multiplier$ combinations of attributes and, for each combination of columns as, we submit to TQ the tasks $(as, 0)$. Each of these tasks informs the worker thread to analyse the combination of columns specified by as and to work on the rows in $[0, BS)$. After that (line 9), the main thread stores in AT the pair (as, BS) to keep track of the next task, which will involve columns as and begin from row BS. Note that we generate more tasks than threads because once a thread returns the result of its analysis to the main thread, there will be a new task waiting to be retrieved from the task queue. This way, the worker threads will not be slowed down by the main thread processing a response.

In line 10, we initialise a counter of terminated threads, which is a variable counting the number of threads that reported to have finished their execution. In line 11, we instantiate a set that will contain the combinations of attributes detected as containing uniques. Then, in lines 12–40 we iterate

```
Input: (i) Dataset $\mathcal{D}$ consisting of $\|\mathcal{R}\|$ records of attributes from $\mathcal{A}$, (ii) number
       of worker threads $t$, and (iii) size of each task in number of rows $BS$
Output: The minimal attributes' combinations containing uniques $\mathcal{U}$
Data: Task queue $TQ$ and result queue $RQ$
1  begin
2      $\mathcal{I}$ = CreateIndex($\mathcal{D}$) ;          // Index creation for $\mathcal{D}$
3      for $i=0;\ i < t;\ ++i$ do
4          SpawnWorkerThread($TQ$, $RQ$, $\mathcal{I}$, $BS$);
       // Initialize the task queue $TQ$ (steps 5-9)
5      $AT = \varnothing$ ;        // Set of analyzed combinations of
                                      attributes and row blocks
6      for $s = 0;\ s < (multiplier * t);\ ++s$ do
7          $as$ = GenerateAttributeSet() ; // Generate a new
                                              set of attributes
8          Push($TQ$, $(as, 0)$) ;    // Push the task $(as, 0)$ in $TQ$
9          Insert($AT$, $(as, BS)$) ;
10     $terminated$ = 0 ;     // Counter of worker threads that
                                 completed their execution
11     $\mathcal{U} = \varnothing$;
12     while $terminated \neq t$ do
13         $result$ = Pop($RQ$); // Pop the result queue; this is
                                     a blocking call
14         if $result.terminated$ then
15             $++terminated$;
16             continue;
17         if $result.unique\_found$ then
18             $\mathcal{U} = \mathcal{U} \cup result.as$;
19             Ban($result.as$); // Notify the attribute set
                                    generator that $as$ contains
                                    uniques
20             $removed$ = 0 ;   // Counter of elements removed
                                    from $AT$
21             foreach $(as, offset) \in AT$ do
22                 if IsSubsetOf($result.as$, $as$) then
23                     remove $(as, offset)$ from $AT$;
24                     $removed$++;
25             for $i = 0;\ i < removed;\ ++i$ do
26                 $as$ = GenerateAttributeSet();
27                 Insert($AT$, $(as, 0)$);
           // Push new tasks to the task queue
28         if $AT \neq \varnothing$ then
29             $(as, offset) \leftarrow AT$;       // Extract the next set
                                                      of attributes
30             Push($TQ$, $(as, offset)$) ; // Send the next batch of
                                               work for attributes $as$
31             if $offset + BS < \|\mathcal{R}\|$ then
32                 Insert($AT$, $(as, offset + BS)$);
33             else
                   // Generate a new set of attributes
34                 if $as'$ = GenerateAttributeSet() then
35                     Insert($AT$, $(as', 0)$);
36         else
37             message = (Terminate);
38             message.terminate = True;
39             for $i = 0;\ i < t;\ ++i$ do
40                 Push($TQ$, $message$) ; // Send the thread the
                                              command to terminate
41     return $\mathcal{U}$;
```

Algorithm 4: FPVI, main thread



(a) Index after row 0

(b) Index after records 1, 2 and 3

(c) Index after row 6

(d) Index of entire dataset

Fig. 3: Evolution of the index structure for the dataset of Table I

until all threads have completed executing all the tasks in TQ and have reported to the main thread via the result queue RQ. Specifically, in line 13, the main thread dequeues a message from RQ. Note that the `Pop` function is blocking, in the sense that it blocks the execution of the main thread until a worker thread has pushed a message to RQ. Also note that *result* is a data structure that contains the following fields: (i) `terminated`: a boolean value to notify that the worker thread terminates its execution, (ii) `unique_found`: a boolean value that is set if the worker thread has found a unique in the assigned task, (iii) `as`, the combination of columns analyzed, and (iv) `r`: the row of `as` where the first unique was found.

In line 14, the main thread checks if the message $result$ is a notification from a worker thread signalling that it terminated. If that is the case, the main thread increments the counter of terminated threads. On the other hand, if the message is not a notification of termination, it will contain the result of the analysis of the given task. In this case, the main thread checks whether the worker thread identified a unique (line 17) and, if so, it adds the combination of attributes defined in the field `as` to set $\mathcal{U}$ (line 18). The main thread also notifies the generator of attribute sets (by calling the `Ban` function) that the combination of columns $result.as$ contains uniques, to prevent the generation of combinations of attributes that are supersets of $result.as$. Subsequently, in lines 20–24, the main thread scans AT searching for combinations of attributes that are supersets of$result.as$ to block them from being analyzed. Each such element is removed from AT (line 23).

Subsequently, for every element removed from AT, a new entry is generated (lines 25–27). Function `GenerateAttributeSet` maintains an internal state that keeps track of the last generated attributes' combination. It is also responsible of returning only *valid* combinations of attributes, i.e., combinations of attributes that are not supersets of banned ones. In line 28, the main thread verifies that there are still combinations of attributes that have to be analyzed by the worker threads. In lines 29–36, new tasks are generated and pushed to the task queue. Our task generation process places preference to analyzing the dataset in a way that all active combinations of columns are analysed for a batch of rows, and the same combinations of columns are subsequently analysed for the next batch of rows, until all rows for the given combinations have been processed. Then, the next attributes' combination is analyzed for uniques. We opted for this approach because it allows the early identification of combinations of columns that contain uniques, thereby reduces the amount of unnecessary work preformed by the worker threads. Once a next task has been generated, it is enqueued to the task queue TQ (line 30).

In lines 31–35, the main thread verifies if the combination of attributes specified by as has still rows to be analyzed. In this case, it stores in AT value "*(as, offset + BS)*", which represents the state of the next valid task for as. On the other hand, if the analysis of as has finished, the main thread retrieves a new valid combination of attributes (line 34). If `GenerateAttributeSet` returns a valid combination of attributes, then the main thread inserts it to AT, as the pair $(as', 0)$, signifying that the task will involve attributes as and start from the first row (line 35). If the test in line 28 does not hold, which means AT is empty, then the main thread needs to notify the worker threads that there are no more tasks to execute. This is done by pushing t termination messages to the task queue. Specifically, the main thread appends the termination messages as last messages of the queue, thus the worker threads will have to read, and execute, all the pending tasks before reading a termination message and terminating their operation (lines 37–40). Last, in line 41, the main thread returns the set $\mathcal{U}$ that contains the distinct combinations of attributes containing uniques.

Algorithm 5 shows the function that is executed by each worker thread. It takes as arguments the task queue TQ, the

```
Input: (i) Task queue $TQ$, (ii) result queue $RQ$, (iii) index $\mathcal{I}$ of dataset $\mathcal{D}$, and
       (iv) size of each task in number of rows $BS$
1  begin
2      while True do
3          response = (terminated, unique_found, $as$, $r$);
4          task = Pop($TQ$);  // Retrieve the next task from $TQ$
5          if task.terminate then
6              response.terminate = True;
7              Push($RQ$, response);    // Notify that the thread
                                           is stopping
8              return;
9          max_row = min{task.offset + $BS$, $\|\mathcal{I}\|$};
10         r = task.offset;
11         for $r <$ max_row; ++r do
12             if IsUnique($\mathcal{I}$, $as$, $r$) then
13                 break;
14         if $r \neq$ max_row then
15             response.unique_found = true; // Notify the main
                                                thread that row $r$ is
                                                unique, w.r.t. $as$
16             response.$as$ = task.$as$;
17             response.r = r;
18         else
19             response.unique_found = False;  // No unique found;
                                                  notify main thread
20         Push($RQ$, response)
```

Algorithm 5: FPVI, worker thread

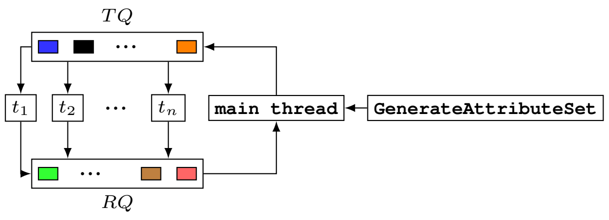


Fig. 4: FPVI: process architecture

result queue RQ, the index $\mathcal{I}$, and the batch size BS. The function immediately tries to retrieve messages from TQ (lines 3–4). Note that the Pop function is blocking, so it blocks the execution of the caller thread if the queue is empty. In line 5, the worker thread checks whether the message extracted from TQ is a notification of termination, in which case it notifies the main thread of its intention to terminate (line 6), pushing the response to the result queue (line 7) and exiting (line 8).

Otherwise, in line 9, the worker thread computes the minimum value between the number of rows in the dataset, which we remind is equivalent to the size of the reverse index $\mathcal{I}$, and the offset value defined in the task message increased by BS. Next, the worker thread checks every row in $[r,$ *max_row*$)$ for uniques with respect to the combination of columns defined in *task.as* (lines 11-13). In line 14, the worker thread checks if a unique was found, in which case it sets the field *unique_found* to true (line 15), the field as to *task.as*, and the field r to r. Otherwise (line 18), the worker thread sets the field *unique_found* to *False* (line 19). In either case, it appends the response to RQ and retrieves a new task from TQ. The architecture of FPVI is shown in Figure 4.

Remark We note that FPVI can be extended to find all minimal combinations of attributes leading to k or less individuals ($k \geq 1$). To do that, one needs to add parameter k to the algorithm and change the IsUnique function (Algorithm 5) so that it returns *True* only if there are at most k records with the same combination of attributes' values.

Complexity analysis The worst-case complexity of the algorithm executed by each worker thread is $O(2^{\|\mathcal{A}\|} \times (\|\mathcal{R}\| \times log\,\|\mathcal{R}\|))$. This cost is divided among the t threads, leading to a complexity of $O(2^{\|\mathcal{A}\|} \times \|\mathcal{R}\|\, log\,\|\mathcal{R}\|/t)$. The overall worst-case computational complexity of FPVI is $O(\|\mathcal{R}\| \times log\,\|\mathcal{R}\| \times \|\mathcal{A}\| + 2^{\|\mathcal{A}\|} \times \|\mathcal{R}\|\|\mathcal{A}\|/BS) = O(2^{\|\mathcal{A}\|} \times \|\mathcal{R}\|\|\mathcal{A}\|/BS)$.

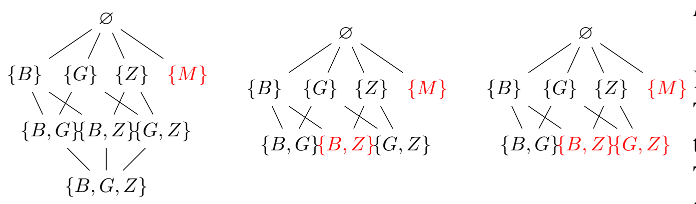
Running example Consider the dataset of Table I and assume that $t = 3$ and $BS = 5$. Upon invocation, the main thread calls Algorithm 2 to build the index and then starts 3 worker threads, passing to each of them a reference to the task queue TQ and to the result queue RQ. Then, it generates the initial batch of tasks. Assuming that the *multiplier* is set to 1.333 (which is close to $\|\mathcal{A}\|/t$), this leads to the generation of 4 tasks. The tasks that are pushed to TQ are: (i) $(\{B\}, 0)$; (ii) $(\{G\}, 0)$; (iii) $(\{Z\}, 0)$; and (iv) $(\{M\}, 0)$. The result queue RQ is originally empty and AT, the set of analyzed combinations of attributes and row blocks, contains the following elements: (i) $(\{B\}, 5)$; (ii) $(\{G\}, 5)$; (iii) $(\{Z\}, 5)$; and (iv) $(\{M\}, 5)$.

Now assume that t_0 retrieves from TQ the task $(\{B\}, 0)$ and tries to identify uniques in the indexes of records 0 to 4. For each record, it checks whether the corresponding entries of the reverse index contain a single (hence unique) element. Thus, it invokes the IsUnique function with parameters ($\mathcal{I}$, $\{B\}$, i), where i is the record id. In this case, IsUnique verifies that the cardinality of the entry $(B, 0)$ of Table II is not 1. Since there are no uniques in the first 5 rows of column R of Table II, t_0 notifies accordingly the main thread, by pushing a response to RQ. The same holds for t_1 and t_2, when processing $(\{G\}, 0)$ and $(\{Z\}, 0)$, respectively.

Immediately after the first thread pushes a response to RQ, the main thread is able to continue its execution by pushing new tasks to TQ. Assume that t_1 is next assigned the execution of task $(\{M\}, 0)$. The thread proceeds to check if there is a unique between lines 0 and 4 of Table II, by using the IsUnique function. Indeed, it detects that the cardinality of the record $(M, 2)$ is 1, which means that the associated value is unique for the dataset. Accordingly, t_1 pushes a response to RQ. The other threads, continue their execution as normal without detecting uniques in the tasks assigned to them.

Once the main thread retrieves from RQ the message from t_1, which has the *unique_found* field set to *True*, it proceeds to ban $\{M\}$ and its supersets. This prunes the elements of the lattice, as shown in Figure 5a. Next, the main thread removes all the entries of AT that are supersets of $\{M\}$. In this case only the entry $(\{M\}, 5)$ of AT is identified as superset of $\{M\}$, and is discarded. This leads to the generation of a new set of attributes, $\{R, B\}$, through the invocation of the GenerateAttributeSet function.

Similarly to what t_1 did during the analysis of $\{M\}$, when a worker thread receives the task $(\{B, Z\}, 0)$ it will find that record 1 is unique and will notify the main thread. The same will happen for task $(\{G, Z\}, 0)$, as record 0 is unique. The exploration of the search space will continue in a similar way with worker threads finding uniques and banning attribute sets, such as $\{B, Z\}$ and $\{G, Z\}$, and their supersets. This causes a



(a) Pruning $\{M\}$ (b) Pruning $\{B, Z\}$ (c) Pruning $\{G, Z\}$

Fig. 5: The pruning operations for the dataset of Table I

further pruning of the search space, leaving only the elements shown in Figure 5c to be explored.

V. Experimental evaluation

In this section we present the experiments that we conducted to evaluate the performance of our proposed algorithm. All experiments were executed on a server running RedHat Enterprise Server 6, with 1TB memory and Intel Xeon E7 4870 2.40GHz CPUs, amounting to a total of 80 CPU cores.

A. Datasets

In our experiments, we considered datasets of varying size and number of uniques. In the following, we provide more information about each dataset used in the experiments.

Adult dataset: The Adult dataset[1] is commonly used to evaluate methods for detecting uniques. It consists of 15 attributes and $48,842$ records.

Household power consumption dataset: This dataset[1] contains measurements about the electricity consumption of households, collected between 2006 and 2010. It consists of 9 attributes and $2,075,259$ records.

PUMS dataset: The PUMS datasets are provided by the U.S. Census Bureau[2]. We merged the datasets for California ($437,869$ records), Idaho ($14,984$ records), Texas ($262,349$ records) and Washington ($72,689$ records), to create the PUMS_All dataset. Due to the variance in the number of attributes reported for each state (and also within the same state), we maintained only those records with at least 16 attributes, and kept the first 16 attributes.

HIGGS dataset: The HIGGS dataset[1] reports on measurements performed during the study of the Higgs' particle. Because HIGGS contained too many uniques, we trimmed the values to 10 decimal digits, which significantly reduced the uniques. We then created 22 datasets from HIGGS, spanning from 1M to 11M records, and considered 20 and 28 (max) attributes.

MiniBooNE dataset: The MiniBooNE dataset[1] contains $130,065$ records and 50 attributes. We created a collection of smaller datasets, spanning from (the first) 1K to 130K records and from (the first) 30 to 50 attributes.

YearPredictionMSD dataset: YearPredictionMSD[1] contains $515,345$ records and 90 attributes. We created a collection of smaller datasets, keeping the number of records constant and varying the number of attributes in $[10-35]$.

[1]https://archive.ics.uci.edu/ml/datasets/
[2]http://www.census.gov/main/www/pums.html

B. Baseline methods: MTS2 and MTUI

The existing methods for identifying uniques (see Section II) are either too slow, or assume specific hardware to execute. To provide a fair comparison of our approach with the state-of-the-art, we implemented two baseline methods, called Multi-Threaded SUDA 2 (MTS2) and Multi-Threaded Uniques Identification (MTUI).

MTS2 is a parallel, iterative version of SUDA2 [13] that can execute in both commodity machines and cluster environments, similarly to FPVI. MTS2 does not convert the recursion to an iteration over a stack, as the different tasks are most likely assigned to different worker threads and should therefore be as independent as possible. On the other hand, MTUI is procedurally similar in spirit to MTS2, but leverages on the same pruning mechanism that is used by FPVI. It can be considered as a simpler version of FPVI.

In the following plots (Figures 6 and 7) we refrain from presenting the values for MTS2, as the execution time of the algorithm is huge when compared with that of MTUI and FPVI. We will, however, discuss the runtimes attained by MTS2 in the description of the experiments.

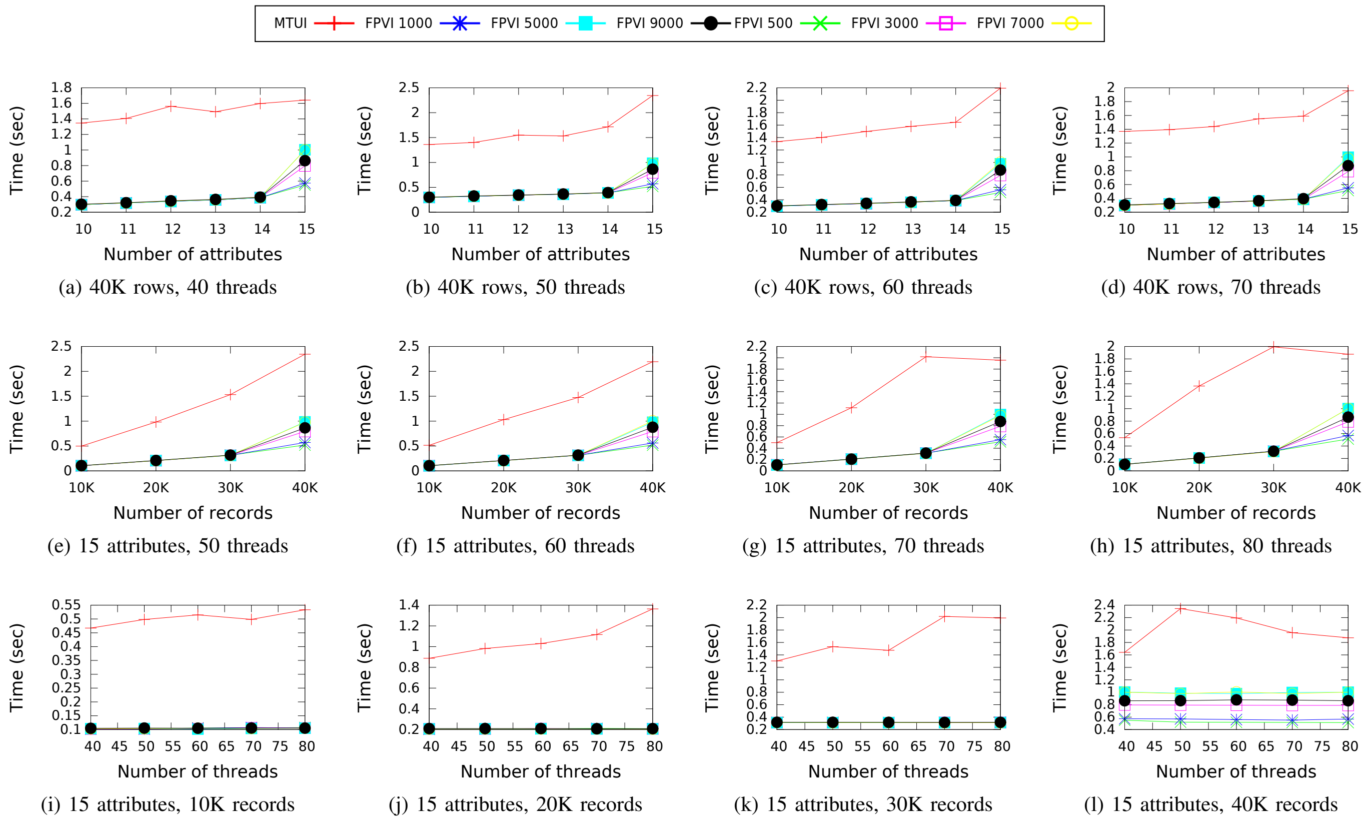
C. Experimental results

We executed many experiments to compare the performance of the FPVI, MTUI, and MTS2 algorithms. Each experiment was executed at least 10 times and the minimum execution time is reported in the graphs. We also executed FPVI with different batch sizes, spanning from 500 to 9000 records.

Figure 6 presents the experimental results that we attained for the `Adult` dataset. To evaluate the scalability of the proposed approach, we created several smaller dataset from the original one, consisting of 10–15 attributes and 10K–40K rows. Figures 6a-d present the performance of the algorithms with respect to a varying number of attributes. All algorithms scale exponentially with respect to this dimension. Note, however, that FPVI scales much better than MTUI, with the execution time for FPVI for 15 attributes being still less than a second, while for MTUI the corresponding time is much higher. For the same series of experiments, the results for MTS2 are on average 1044.2 and 1171.81 seconds, values obtained using 80 and 40 threads, respectively.

Next, Figures 6e-h present the behavior of the algorithms with respect to the number of records, which is linear as also confirmed by our theoretical analysis. Nevertheless, FPVI outperforms MTUI as it scales with a smoother slope. In these experiments, MTS2 requires significantly more time to process the dataset. As an example, in Figure 6e, for 40K records the MTS2 algorithm requires 1099.31 seconds to execute, MTUI requires 1.52 seconds and FPVI only 0.52 seconds with a batch size of 500 (and 0.98 seconds with a batch size of 5000).

Figures 6i-l present the scalability of the algorithms with respect to the number of threads. In these experiments we may immediately notice the limitation of MTUI. In particular, we notice that all the configurations of FPVI present very similar execution times, independently on the number of threads used. This is caused by the fact that the algorithm has reached the



(a) 40K rows, 40 threads (b) 40K rows, 50 threads (c) 40K rows, 60 threads (d) 40K rows, 70 threads

(e) 15 attributes, 50 threads (f) 15 attributes, 60 threads (g) 15 attributes, 70 threads (h) 15 attributes, 80 threads

(i) 15 attributes, 10K records (j) 15 attributes, 20K records (k) 15 attributes, 30K records (l) 15 attributes, 40K records

Fig. 6: Performance of the algorithms on the Adult dataset

minimum time it can achieve for this dataset when using 40 threads. Thus, increasing the number of threads is not going to provide additional gain in performance. This is because the dataset is fairly small from the point of view of FPVI. On the other hand, the execution time of MTUI is increasing with an increase in the number of available threads. This is because the dataset is small and the different threads are interfering with each other. This is more evident in Figure 6l. Specifically, this is caused by the naïve parallelization technique used by MTUI. On the other hand, FPVI uses an advanced parallelization technique that allows the worker threads to operate almost independently, with minimal interference.

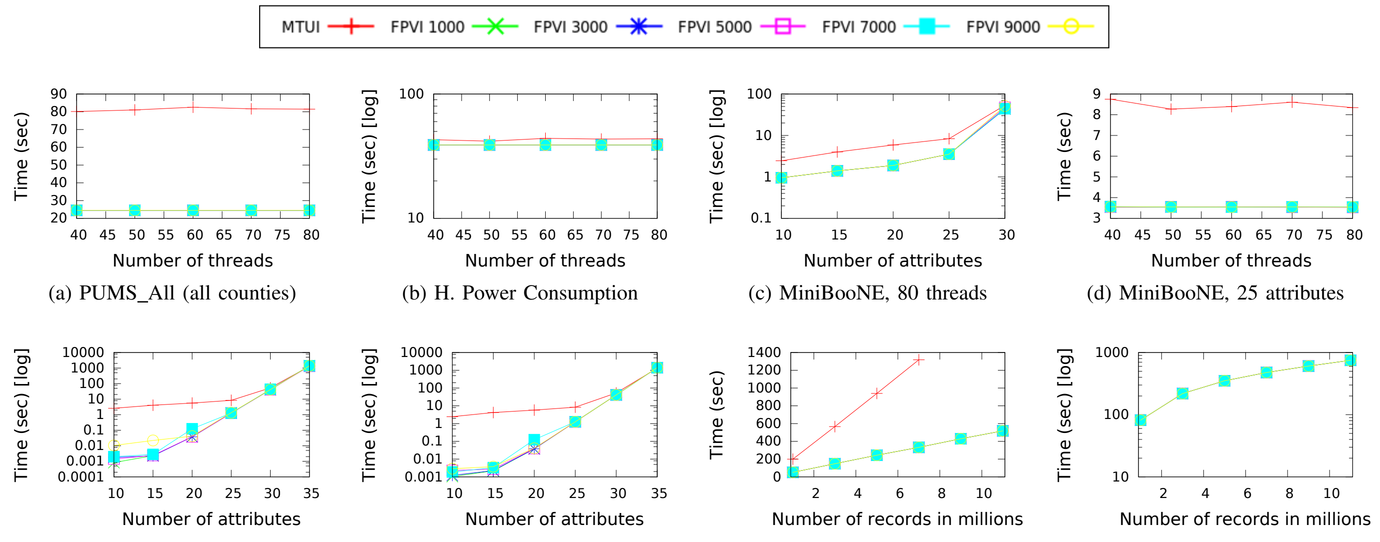
Figure 7 presents the experiments executed on larger datasets than `Adult`, with various characteristics. MTS2 failed to complete in most of the tested cases, when executed with a 2 hours timeout, and when it did finish its runtime was very high compared to that of MTUI and FPVI. Figure 7a shows the execution time for the `PUMS_All` dataset. For this dataset both FPVI and MTUI reached their minimum runtime before 40 threads, with FPVI being faster. Specifically, FPVI required (on average) 24.51 seconds to execute, while MTUI took at least 80.13 seconds. Similarly, in the `Household Power Consumption` dataset (Figure 7b), MTUI took 41.74 seconds, while FPVI took 38.93 seconds. For this series of experiments, MTS2 completed in 795.1 seconds (on average).

Figures 7c and 7d present experiments on `MiniBooNE`. In these figures, we see that the execution time of MTUI and FPVI appears to converge when the number of attributes increases. This is because the inclusion of new attributes from `MiniBooNE` causes the algorithms to find less uniques and, therefore, reduces the impact of the pruning technique.

A similar behaviour appears in Figures 7e-f, where the results for the `YearPredictionMSD` dataset are presented. These figures show a comparable behaviour with 50 and 60 worker threads. For both MTUI and FPVI (with various batch sizes), the execution time increases with an increase in the number of attributes, but FPVI outperforms MTUI.

The last two plots of Figure 7, show the behaviour of the algorithms on `HIGGS`. In both figures we see that FPVI is able to handle large datasets, containing several millions of records, with a small runtime cost. We remind that increasing the number of records decreases the probability that an attribute-value combination is unique in the dataset. MTUI requires significantly more time than FPVI to process the data, as shown in Figure 7g. The behavior of FPVI can be observed better in Figure 7h, where we removed the results for MTUI.

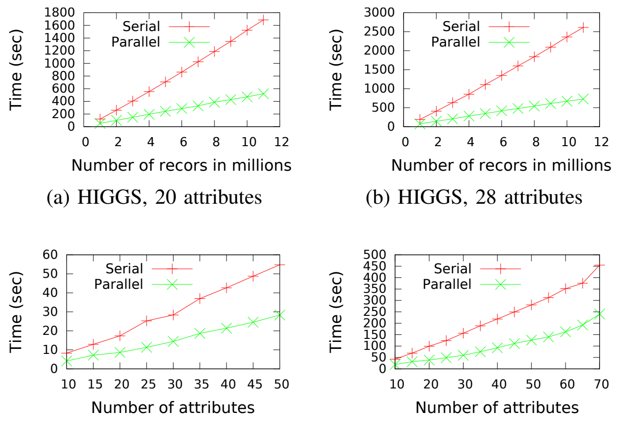
Last, in Figure 8 we present the performance of the parallel indexing approach, implemented in FPVI, versus its serial counterpart, when applied on different datasets. One may immediately observe how faster the parallel implementation is in indexing the data. In particular, the fact that the number of worker threads in the parallel implementation is equal to the number of attributes, reduces the impact of this dimension to the runtime. This way, the FPVI algorithm is much more scalable and able to handle larger datasets.



(a) PUMS_All (all counties) (b) H. Power Consumption (c) MiniBooNE, 80 threads (d) MiniBooNE, 25 attributes

(e) YearPredictionMSD, 50 threads (f) YearPredictionMSD, 60 threads (g) HIGGS, 20 attributes, 70 threads (h) HIGGS, 28 attributes, 70 threads

Fig. 7: Performance of the algorithms on the remaining datasets



(a) HIGGS, 20 attributes (b) HIGGS, 28 attributes

(c) MiniBooNE (d) YearPred

Fig. 8: Create index

VI. Conclusion

In this paper we introduced FPVI, a fast algorithm for identifying minimal sets of attributes that can act as quasi-identifiers, leading to privacy attacks in microdata sets. FPVI operates in a multi-threaded fashion to index the data and scan different attributes' combinations for uniques. Through experiments conducted over several real-world datasets, we showed that FPVI outperforms the state-of-the-art, being able to analyze datasets consisting of millions of records and tens of attributes, in a few minutes.

References

[1] G. T. Duncan, M. Elliot, and J.-J. Salazar-Gonzalez, *Statistical Confidentiality: Principles and Practice*. Springer, 2011.

[2] A. Ramachandran, L. Singh, E. Porter, and F. Nagle, "Exploring re-identification risks in public domains," in *Proceedings of the International Conference on Privacy, Security and Trust*, 2012, pp. 35–42.

[3] L. Sweeney, "K-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 5, pp. 557–570, 2002.

[4] B. C. M. Fung, K. Wang, R. Chen, and P. S. Yu, "Privacy-preserving data publishing: A survey of recent developments," *ACM Computing Surveys*, vol. 42, no. 4, pp. 1–53, 2010.

[5] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008, pp. 111–125.

[6] C. Dwork, "Differential privacy," in *Lecture Notes in Computer Science, Volume 4052, Automata, Languages and Programming*, 2006, pp. 1–12.

[7] J. Gardner, L. Xiong, Y. Xiao, J. Gao, A. R. Post, X. Jiang, and L. Ohno-Machado, "SHARE: System design and case studies for statistical health information release," *Journal of the American Medical Informatics Association*, vol. 20, 2013.

[8] S. E. Fienberg and U. E. Makov, "Confidentiality, uniqueness, and disclosure limitation for categorical data," *Journal of Official Statistics*, vol. 14, no. 1, pp. 385–397, 1998.

[9] D. Lambert, "Measures of disclosure risk and harm," *Journal of Official Statistics*, vol. 9, no. 1, pp. 313–331, 1993.

[10] S. M. Samuels, "A bayesian, species-sampling-inspired approach to the uniques problem in microdata disclosure risk assessment," *Journal of Official Statistics*, vol. 14, no. 4, pp. 373–383, 1998.

[11] A. Takemura, "Minimum unsafe and maximum safe sets of variables for disclosure risk assessment of individual records in a microdata set," *Journal of the Japan Statistical Society*, vol. 32, pp. 107–117, 2002.

[12] M. J. Elliot, A. Manning, K. Mayes, J. Gurd, and M. Bane, "SUDA: A program for detecting special uniques," in *Proceedings of UNECE Work Session on Statistical Data Confidentiality*, November 2005.

[13] A. M. Manning and D. J. Haglin, "A new algorithm for finding minimal sample uniques for use in statistical disclosure assessment," in *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM)*, 2005, pp. 8–15.

[14] P. Yiapanis, D. J. Haglin, A. M. Manning, K. Mayes, and J. Keane, "Variable-grain and dynamic work generation for minimal unique item-set mining," in *Proceedings of the IEEE International Conference on Cluster Computing*, 33–41, p. 2008.

[15] D. J. Haglin, K. R. Mayes, A. M. Manning, J. Feo, J. R. Gurd, M. Elliot, and J. A. Keane, "Factors affecting the performance of parallel mining of minimal unique itemsets on diverse architectures," *Journal of Concurrency and Computation: Practice and Experience*, vol. 21, no. 9, pp. 1131–1158, 2009.

[16] R. Motwani and Y. Xu, "Efficient algorithms for masking and finding quasi-identifiers," in *Proceedings of the Conference on Very Large Data Bases (VLDB)*, 2007, pp. 83–93.

[17] L. Cagliero and P. Garza, "Infrequent weighted itemset mining using frequent pattern growth," *IEEE Transactions on Knowledge & Data Engineering*, vol. 26, no. 4, pp. 903–915, 2014.

[18] D. J. Haglin and A. M. Manning, "On minimal infrequent itemset mining," in *Proceedings of the International Conference on Data Mining (DMIN)*, 2007, pp. 141–147.

[19] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, pp. 487–499.