Persistent Pointer Information

Xiao Xiao Qirun Zhang Jinguo Zhou Charles Zhang

The Prism Research Group
The Hong Kong University of Science and Technology
{richardxx, qrzhang, andyzhou, charlesz}@cse.ust.hk

Abstract

Pointer information, indispensable for static analysis tools, is expensive to compute and query. We provide a query-efficient persistence technique, Pestrie, to mitigate the costly computation and slow querying of precise pointer information. Leveraging equivalence and hub properties, Pestrie can compress pointer information and answers pointer related queries very efficiently. The experiment shows that Pestrie produces 10.5× and 17.5× smaller persistent files than the traditional bitmap and BDD encodings. Meanwhile, Pestrie is 2.9× to 123.6× faster than traditional demand-driven approaches for serving points-to related queries.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis

General Terms Algorithms, Languages, Performance

Keywords Pestrie, fast querying, compact indexing

1. Introduction

Pointer information (points-to and aliasing) is a prerequisite for many static analysis tools. In spite of recent progress [15, 19, 21, 32, 40, 46, 47], obtaining precise pointer information (*i.e.*, context- or flow-sensitive) is still expensive for large-scale production software, which dramatically affects the quality of static analysis in practice.

We observe that, in many scenarios, pointer analysis is performed repeatedly for unchanged code. Hence, persisting and reusing the precise pointer information across the life cycles of static analyses can significantly boost programmer's productivity. We illustrate this benefit with two real-world settings.

1. Consider in-house regression analysis against a code base tagged for a release. First, the persistent pointer information of the released code can be leveraged for change impact analysis [11] and postmortem debugging of field failures [22] that may occur frequently. In addition, several analyses can be pipelined together to carry out a more sophisticated task. For instance, when a memory leak detector [36] is used together with a race detector [25], the persisted pointer information could be shared among different analysis stages to further speed up the overall bug detection tasks.

PLDI'14, June 9 - 11, 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594314

Query	Description
IsAlias(p, q)	Decide if the pointer p is an alias of q
ListPointsTo(p)	Output the points-to set for pointer p
ListPointedBy(o)	Output the pointers that point to memory o
ListAliases(p)	Output the pointers that are aliased to pointer p

Table 1. Queries supported by persistent pointer information.

2. There is an emerging interest in pre-analyzing libraries to scale whole program analysis. Recent work considers encoding the program analysis results of the full JDK library for efficiently performing IDE data flow analysis [31] and quickly generating call graphs for client analysis [2]. In conjunction with fragment analysis techniques [29, 30], persistence techniques can also avoid the duplicated analysis effort on libraries, by separately computing and persisting the points-to relations of a library that are independent of clients.

The usefulness of persistence is multi-fold. But pointer information is typically very difficult to compactly store and, meanwhile, to efficiently interpret, as also observed by other researchers [3]. A good persistence scheme needs to compress gigabytes of pointer information while retaining the ability to efficiently answer pointer related queries. More specifically, it is important to efficiently answer the common queries described in Table 1. The first two queries, **IsAlias** and **ListPointsTo**, are the *de facto* standard for points-to analyses and, of course, should be served efficiently. The query **ListPointedBy** is particularly useful in value-flow analysis [10] and type-state verification [24]. The last query, **ListAliases**, does not widely appear in the literature, but it could be very useful for applications that track the global information flow and look for all aliased pointers of a querying pointer [35, 37, 43]. For example, using the **ListAliases** query to generate the aliasing pairs for a data race detector [25] is 123.6× times faster than the approach described in the original paper (Section 7.1).

We can tackle the storage challenge with an off-the-shelf compressing technique such as bzip. However, general compressing algorithms cannot leverage any semantics of the points-to relation and often still produce hundreds of megabytes. More importantly, the compressed pointer information is not *query-efficient* due to the prolonged decoding process. The state-of-the-art pointer analyses, resorting to binary decision diagrams (BDD) [39] or equivalent context merging (EPA) [41], also cannot simultaneously maintaining high compression and fast querying capabilities. First, BDDs cannot compress the pointer information with heap cloning well, as observed by many researchers [5, 40, 41]. Second, BDDs are *inefficient* for answering queries such as the ones in Table 1. For example, our experiments show that a BDD takes 432.3 seconds to answer the **ListPointsTo** query even for our smallest benchmark antlr (Section 7). This implies that answering the **IsAlias** query using BDDs will be much slower than the use of bitmaps. This is because, given two pointers, we have to first decode their points-

to sets from the BDDs and then intersect the sets to determine if they are aliased. Even worse, to answer the query **ListAliases(p)**, we have to repeat this costly process for all other pointers. In fact, many popular analysis infrastructures, *e.g.*, LLVM, GCC, Open64, Soot, and WALA, rarely employ BDDs to store pointer information [33]. EPA suffers from the same querying problem that requires the decompression of the encoded information.

In this paper, we show two observations that can be exploited to generate compact and query-efficient persistent pointer information. The first observation, referred to as the *equivalence* property, is based on the conventional observation of equivalent pointers and objects [14]. Previous work mainly focuses on detecting the equivalent pointers and objects *before and during* the points-to analysis [13, 28]. We study a wide range of programs and show that, *after* the points-to analysis, even for the precise flow-sensitive and context-sensitive ones with heap cloning, there is a large number of equivalent sets in the points-to information.

The second observation is the *hub* property, referring to objects that frequently appear in the intersection of two points-to sets, playing the role of *hubs* in the aliasing relations. Hubs can be leveraged to reduce storage size and improve the query efficiency because pointers that point to the same hub are implicit alias pairs. We assign a *hub degree* to every object and observe that the distribution of the hub degrees for all objects produced by the same points-to algorithm are similar in spite of the different programs used. This suggests that the hub property is not an accidental phenomenon and is caused by the imprecise nature of the points-to algorithms.

We propose the Pestrie encoding technique that leverages both the equivalence and the hub properties. The key idea is to partition the pointers and to divide the aliased pairs into two classes: the *internal* pairs and *cross* pairs. An internal pair consists of two pointers assigned to the same partition, which can be decided by comparing partition IDs. A cross pair is formed by two pointers from different partitions and encoded using *rectangular formulas*. The effectiveness of Pestrie is guaranteed by the partition strategy based on hub degrees, which aims to maximize the number of internal pairs and minimize the number of cross pairs.

We experimentally compare Pestrie to traditional encoding techniques based on bitmaps, BDDs, and bzip using a set of C and Java programs. Quantitatively, Pestrie generates the smallest encoding, 10.5× and 17.5× smaller than those produced by the bitmap and the BDD techniques. Pestrie is also 2.9× faster in computing aliasing pairs than a traditional demand-driven approach for answering the **IsAlias** query [25], and it is 123.6× faster than the demand-driven approach if the novel **ListAliases** query is used. In addition, constructing and loading the Pestrie encoding only takes few seconds. To sum up, our work makes three contributions:

• *Empirical study*: We investigate the characteristics of pointer information and characterize the equivalence and hub properties.

• Pestrie *encoding*: We propose an encoding scheme that leverages both the equivalence and the hub properties to compact aliasing relations and process queries efficiently.

• *Large scale experiments*: We compare Pestrie encoding scheme to bitmap, BDD, and bzip, to demonstrate its compaction superiority and querying efficiency on a set of C and Java programs.

We organize the rest of this paper as follows. We first present our empirical results for pointer information in Section 2. Next, we describe the Pestrie encoding methodology in Sections 3-6. Finally, we show the experiments in Section 7, discuss the related work in Section 8, and conclude our paper in Section 9.

2. Characteristics Study

We first present an empirical study to show the equivalence and the hub characteristics of points-to relations that are *common in spite of the differences in the programming languages and the points-to algorithms*. We study a set of C and Java benchmarking programs (Table 2) widely used in the points-to analysis research literature [5, 13] and divide the subjects into three groups. We extract the points-to information of the programs in the first group with the flow-sensitive algorithm by Lhoták *et al*. [19]. The programs in the second group are selected from Dacapo-2006 and processed by the 1-object-sensitive analysis with heap cloning in Paddle [20]. We use JDK 1.4 library for the points-to analysis because this BDD-based analysis takes too much time on higher versions of JDKs. The last group of programs are selected from the Dacapo-9.12 suite. We resolve the use of reflection in these programs by Tamiflex [4] using the *default* input and analyze them with the JDK 1.6 library using geometric points-to analysis [40].

Program	Language	LOC	#Pointers	#Objects
samba	C	2112.7K	1004880	237201
gs	C	1508.1K	711082	150009
php	C	1312.4K	673156	146760
postgreSQL	C	1189.2K	584774	131886
antlr	Java	75.4K	302560	76970
luindex	Java	67.4K	269878	70426
bloat	Java	188.4K	625056	129471
chart	Java	375.1K	890971	234811
batik	Java	404.5K	766238	137488
sunflow	Java	326.2K	552974	106456
tomcat	Java	357.5K	657394	103627
fop	Java	415.1K	1173406	201122

Table 2. Characterization of the benchmark. **LOC** of C program is the number of instructions of its LLVM bitcode file. For Java program, it is the number of Jimple instructions [38]. The columns **#Pointers** and **#Objects** are the number of pointers and objects.

We normalize the pointer information to a matrix representation. The points-to matrix PM is a binary matrix, where $PM[i][j] = 1$ means the pointer i may point to the object j. Although the normalized points-to representation is simple, we can show that the output of most flow-sensitive, path-sensitive, and context-sensitive algorithms [12, 39–41] can be transformed to the matrix format without any loss of precision. We describe the transformation methodology in Section 6.

2.1 Equivalence Property

Figure 1 presents the summary of percentage of non-equivalent pointers and objects in our subjects. We consider two pointers to be equivalent if their points-to sets are the same. Similarly, two objects are considered equivalent if they are pointed by the same set of pointers. On average, the number of pointer equivalence classes is 18.5% of the number of pointers. The objects are more diverse, with the number of equivalence classes being 83% of the number of objects. This result tells us that, although we have used precise points-to algorithms, there are a large number of undetected equivalent pointers in the final points-to result. The high degree of equivalence also explains the small memory consumption of BDD-based points-to analysis because a BDD can detect and merge all equivalences *online*.

Nevertheless, we can merge the equivalent pointers and objects by using a sparse bitmap, which generates more compact information compared to BDDs. In general, every BDD node contains rich meta-data for structural maintenance. For example, in buddy[1] and JavaBDD[2], every BDD node occupies 20 bytes [5]. For a large program with precise pointer information, a BDD can easily grow to millions of nodes, which requires large storage space for the meta-data [5]. In contrast, the meta-data overhead for the bitmap is much

[1] `http://buddy.sourceforge.net/`

[2] `http://javabdd.sourceforge.net/`

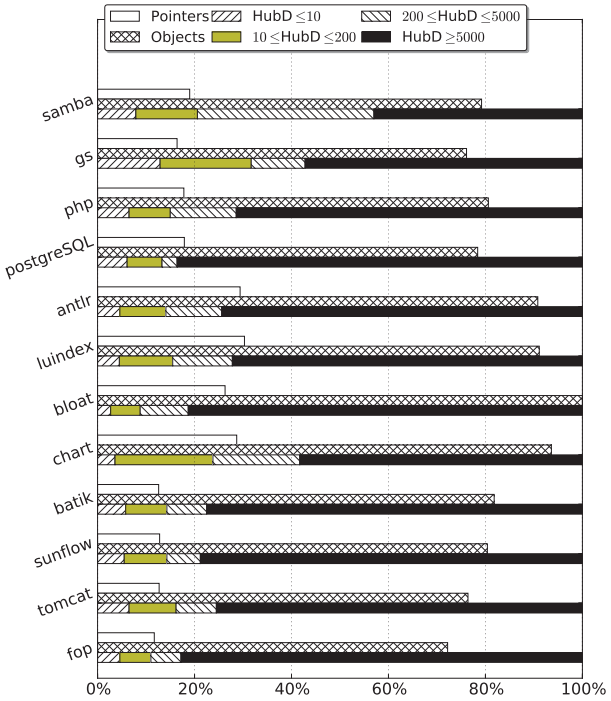

Figure 1. Percentage of non-equivalent pointers and objects, and the distribution of hub degrees.

smaller. Moreover, with bitmaps, we can calculate matrix multiplications quickly. Therefore, we can efficiently generate the alias matrix AM by computing $PM \times PM^T$ to support the **IsAlias** and **ListAliases** queries.

However, using bitmaps to compress the alias matrix AM is not effective because AM does not have a large number of equivalent rows. Since the alias matrix is critical to efficiently answer the **IsAlias** and **ListAliases** queries, we need a more effective way to compress it.

2.2 Hub Property

When intersecting the points-to sets of pointers p and q to determine if they are aliases, we observe that certain objects frequently appear in the intersection sets of two pointers, playing the role of *hubs* in the alias relations. Intuitively, an object pointed by many pointers is more likely to be a hub. Formally, we define the *hub degree* metric as follows.

Definition 1. *Given an object o and its pointed-by set $PM^T[o]$, the hub degree H_o of o is:*

$$H_o = \sqrt{\sum_{p \in PM^T[o]} (|PM[p]|)^2}$$

where $|PM[p]|$ is the points-to set size of p

If we treat the points-to relations as a bipartite graph, our hub degree formula is equal to the two-round iteration of the hub score formula in the HITS algorithm [16]. We could also simply define the hub degree as $\left|\mathrm{PM}^T[o]\right|$, *i.e.* the number of pointers that point to the object o. However, with this metric, we cannot distinguish two objects that are pointed by the same number of pointers. We will justify the design of our hub degree metric in Section 5.2.

The distribution of the hub degrees of our benchmark programs is also plotted in Figure 1. On average, the hub degrees of 70.2% of the objects are larger than 5000. This means large hub nodes frequently appear in points-to results. In fact, Das *et al.* have a similar observation called *blob* nodes in designing their one-level-flow algorithm [8].

An interesting phenomenon is that the programs processed by the same points-to algorithm have similar percentages of the pointers and objects equivalence classes as well as the hub degree distributions. This tells us that *the existence of large numbers of equivalent pointers and hub nodes is due to the imprecise nature of the points-to algorithm and not the result of the code patterns in our selected programs*.

PM						PM^T							
	o_1	o_2	o_3	o_4	o_5		p_1	p_2	p_3	p_4	p_5	p_6	p_7
p_1	1	0	0	0	1	o_1	1	1	1	1	0	0	0
p_2	1	0	0	0	0	o_2	0	0	1	1	0	1	0
p_3	1	1	1	0	1	o_3	0	0	1	1	0	0	1
p_4	1	1	1	1	0	o_4	0	0	0	1	1	0	0
p_5	0	0	0	1	0	o_5	1	0	1	0	0	0	1
p_6	0	1	0	0	0								
p_7	0	0	1	0	1								

Table 3. Sample points-to matrix and its transpose form.

3. Pestrie Persistence Scheme

In this section, we present the Pestrie scheme that persists pointer information by utilizing both the equivalence and the hub properties.

3.1 Constructing Pestrie Representation

The main idea for the Pestrie construction is partitioning all pointers so that the equivalent pointers are explicitly represented to allow points-to information be retrieved by a reachability analysis. A naïve approach is to pick an arbitrary object o_1 and partition the pointers into two groups, one containing all pointers pointing to o_1 and the other for the rest of pointers. For each group, we use the next object o_2 to partition them into two groups again. We recursively partition the groups with objects $o_3, \ldots, o_m$, for all objects. In the end, two pointers in the same group must have identical points-to sets. The partition process can be visualized as a *decision tree*. Pestrie is the compact form of this decision tree.

We can construct Pestrie more efficiently. We illustrate the construction algorithm with an exemplary points-to matrix in Table 3. We first compute the pointed-by matrix PM^T. We then sort the objects by their hub degrees to establish the *object order*. In our example, the object order is o_1, o_2, o_3, o_4, o_5. We further process the rows of PM^T in the object order by scanning the pointers in each row. The result of each step is illustrated in Table 4. We further explain each step as follows.

Step 1. We process row o_1 of PM^T and put all pointers that point to o_1, together with o_1, into group-1.

Step 2. We process row o_2 and identify a new pointer p_6, which has not yet been added to the Pestrie. We put o_2 and p_6 into a new group, namely, group-2. For the other two pointers p_3 and p_4 in row o_2, we extract them out of group-1 and put them in a new group-3. We introduce two edges to encode the information that p_3 and p_4 point to both o_1 and o_2. Specifically, we insert the *tree edge* (solid arrow) *group-1* $\rightarrow$ *group-3* since the members of group-3 are the previous members of group-1. Hence, group-3 is an offspring of group-1. Moreover, we insert the *cross edge* (dashed arrow) *group-2* $\rightarrow$ *group-3* representing the fact that the two groups have no prior connection.

Step 3. We repeat step 2 to process the pointers in row o_3. This time, group-3 becomes empty because p_3 and p_4 are again pulled out. Since it is unnecessary to produce an empty group, we keep p_3 and p_4 in group-3 and directly connect group-4 to group-3 by a cross edge. However, as we will see, directly keeping p_3 and p_4 in group-3 yields an incorrect points-to relation.

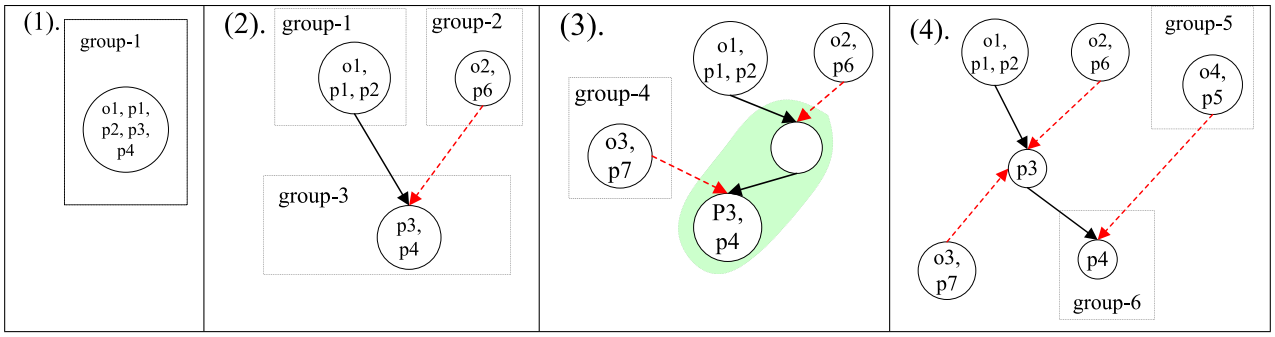

Table 4. Partitioning process. In each step we take a row of PM^T and assign the pointers in that row to some groups.

Step 4. & 5. Again, we repeat step 2 to process rows o_4 and o_5. During the processing of row o_4, we extract p_4 from group-3 and form a new group, group-5, containing o_4 and p_5. Similarly, during the processing of row o_5, we extract p_1, p_3 and p_7 to form a new group containing o_5. The final Pestrie is given in Figure 2, where the edge labels will be explained shortly.

This partitioning algorithm works in $O(nm)$ time, where n and m are the numbers of pointers and objects. Specifically, scanning the input matrix takes $O(nm)$ time, and moving pointers among different groups is also $O(nm)$, since every pointer is moved at most m times and there are n pointers. Pestrie has $O(n + m)$ groups and $O(nm)$ cross edges. Hence, the space complexity is $O(nm)$. Nevertheless, due to our object order defined by the hub degree, the memory consumption of Pestrie in practice is very small. We offer more discussions on this issue in Section 5.2.

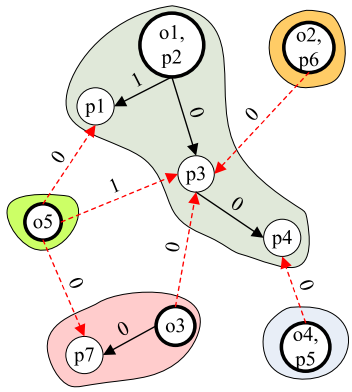

Figure 2. Pestrie for our sample points-to matrix. Edge labels are used in ξ-reachability analysis.

3.2 Pestrie Properties

According to the construction, each Pestrie node represents a group of pointers that have identical points-to sets. Therefore, each Pestrie node forms an *equivalent set (ES)*. For convenience, we use the terms node and group interchangeably. In particular, we define a Pestrie node as an *origin* iff it contains the object o. Two Pestrie nodes are connected via either a cross edge or a tree edge iff they contain pointers that point to some common objects. Moreover, the Pestrie nodes connected via tree edges form a *Partially Equivalent Set (PES)* since they point to a common origin but their points-to sets are not fully identical.

The internal structure of a *PES* is a tree, because every group is built by extracting members from its parent, an operation that cannot produce cycles. An origin is the root of the tree since all groups are extracted from it and, hence, every *PES* has a unique origin. Due to the uniqueness, we use the origin as the *PES* identifier, such as *PES* o_1. Two pointers form an *internal pair* iff they belong to the same *PES*. We associate each pointer in the Pestrie with its *PES* identifier. It is immediate two pointers are aliases if they have the same *PES* identifiers. We use an example to summarize the Pestrie terminology.

Example 1. *Consider a* Pestrie *in Figure 2. There are nine equivalent sets (*ES*s) depicted as* Pestrie *nodes. The shadowed areas represent five partial equivalent sets (*PES*s). In each of the* PES*, the bold node represents the origin. Tree edges in* Pestrie *are depicted as solid arrows and cross edges are dashed arrows. Moreover, (p_3, p_4) is an internal pair.*

The most important Pestrie property is that points-to information could be retrieved by a reachability analysis.

Lemma 1. *A pointer p pointing to object o_j implies that p is reachable from o_j in the* Pestrie.

Proof. We prove it by a case analysis.

Case 1. The *PES* identifier of p is o_j. By the Pestrie construction, p is reachable from its origin o_j.

Case 2. The *PES* identifier of p is o_i, where $i \neq j$. In this case, p is connected to o_j via a path that consists of a cross edge. The reason is that, when we use an object o_j for partition, we connect cross edges to the the set of Pestrie nodes T that are not in *PES* o_j. Since we have $p \in T$, p is reachable from o_j via a cross edge $o_j \rightarrow n_p$. Although p may be moved to other groups in the subsequent construction steps, p is always reachable from o_j through $o_j \rightsquigarrow n_p \rightsquigarrow p$, where $n_p \rightsquigarrow p$ describes the ancestor to descendant path within *PES* o_i.

□

However, p_i is reachable from o_j does not imply p_i points to o_j. Consider Figure 2 again. There is a path between p_4 and o_5, but p_4 does not point to o_5. The problem is that *the cross edge $o_5 \rightarrow p_3$ is built after we isolate p_3 and p_4 in* **Step 4**. If we permit empty group, p_3 is moved to a child group in **Step 5** and we create two tree edges $p_\emptyset \rightarrow p_3$ and $p_\emptyset \rightarrow p_4$, where $p_\emptyset$ is the original group of p_3. Therefore, the path between p_4 and o_5 is $p_4 \leftarrow p_\emptyset \rightarrow p_3 \leftarrow o_5$, and consequently, p_4 is unreachable from o_5. Since Pestrie does not permit empty groups, we need to handle carefully the hidden empty groups $p_\emptyset$ and their corresponding tree edges in the reachability analysis. Namely, if a path starts with a cross edge $x \rightarrow y$, it can only go through the tree edges $y \rightarrow z$ under certain conditions. The conditional reachability analysis is called ξ-reachability, which can correctly retrieve the points-to information from Pestrie.

3.3 ξ-Reachability

The ξ-reachability analysis requires annotations on both the tree and cross edges. For convenience, we call a tree edge $y \rightarrow z$ a tree edge of y. Suppose node y already has k ($k \geq 0$) tree edges. When adding a new tree edge $y \rightarrow z_1$, we label it by k indicating that it is the $(k + 1)^{th}$ tree edge of y. When building a cross edge $x \rightarrow y$,

we label it by the number of tree edges of y at that time and the label on the cross edge is called ξ-*value*. With the labels, we define that the node u is ξ-reachable from the node x iff there is a path $x \xrightarrow{\omega_1} y \xrightarrow{\omega_2} z \rightarrow \ldots u$, which satisfies:

1. $x \xrightarrow{\omega_1} y$ is a cross edge and the edges $y \xrightarrow{\omega_2} z \rightarrow \ldots u$ are all tree edges;
2. $\omega_2 \geq \omega_1$.

We refer to the second property $\omega_2 \geq \omega_1$ as the ξ-condition. The condition guarantees that all tree edges on the path are created after the creation of the cross edge $x \rightarrow y$. Since the tree edge $y \rightarrow z$ is created earlier than any of the tree edges in the sub-tree of z, the ξ-condition only concerns the labels on the first two edges. In consequence, we have the following theorem:

Theorem 1. *A pointer p_i points to an object o_j iff p_i is ξ-reachable from o_j in the* Pestrie.

Example 2. *Consider the example in Figure 2. The tree edge $p_3 \rightarrow p_4$ created in* **Step 4** *is the 0^{th} edge, hence the cross edge $o_5 \rightarrow p_3$ created in* **Step 5** *is labeled by* 1. *Therefore, p_4 does not point to o_5, since the path $o_5 \xrightarrow{1} p_3 \xrightarrow{0} p_4$ is not a ξ-path.*

3.4 Generating Pestrie Persistent File

Our aim of building Pestrie is to answer alias queries efficiently. However, up until this step, the alias information can only be retrieved by the ξ-reachability analysis. Consider the **IsAlias(p, q)** query. If pointers p and q are in the same *PES*, we can quickly decide that they are aliases since (p, q) is an internal pair. Otherwise, we check whether or not (p, q) is a *cross pair* by testing whether p and q are ξ-reachable from an object o simultaneously.

However, using ξ-reachability to check cross pairs takes $O(n + m)$ time, which is slow. A better approach is to encode cross pairs and to look-up the encoded result at query time. A naïve way to generate all cross pairs is enumerating all objects o and pairing up all nodes that are ξ-reachable from o. Instead, since the ξ-reachable nodes induced by a cross edge form a sub-tree, we employ a *sub-tree pairing* approach to efficiently generate cross pairs. This encoding technique, referred to as *rectangle encoding*, compactly represents the cross pairs and achieves $O(\log n)$ query time.

3.4.1 Encoding Cross Pairs

The insight of efficiently generating cross pairs is that the set of ξ-reachable nodes with respect to a cross edge $x \xrightarrow{\omega} y$ is a sub-tree of node y. For the example in Figure 3, node y has $(\omega + k + 1)$ children. Among them, only some children z with tree edges $y \xrightarrow{\omega'} z$ are ξ-reachable from x, where $\omega' \in [\omega, \omega + k]$. These children form a sub-tree of y, which is highlighted in the circled area.

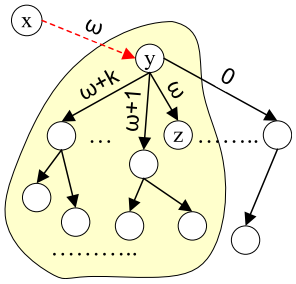

Figure 3. Encoding ξ-reachable nodes. If the pre-order of y is I_y and the largest pre-order of z is E_z, we encode the circled sub-tree as $[I_y, E_z]$.

We extend the interval labeling scheme [1] to compactly represent sub-trees. We first associate each tree node n with an interval label $[I_n, E_n]$, where I_n is the pre-order timestamp obtained from the DFS walk on the tree and E_n is the largest pre-order timestamp of the whole sub-tree rooted at n. To obtain the labels I_n and E_n, we perform a DFS traversal on Pestrie that excludes the cross edges. We visit the *PES* in object order of their origins. For a node that is not an origin, we follow its tree edges in reversed order, *i.e.* the k^{th} tree edge is visited before the $(k-1)^{th}$ tree edge. For an origin, it is free to visit its tree edges in any order, since a ξ-path cannot pass an origin. The result of the DFS traversal is assigning an interval label to every node.

	o_1, p_2	p_3	p_4	p_1	$o_2, p6$	o_3	$p7$	$o_4, p5$	o_5
I	0	1	2	3	4	5	6	7	8
E	3	2	2	3	4	6	6	7	8

Table 5. The pre-order and largest pre-order timestamps for all nodes.

The interval labels for our sample Pestrie (Figure 2) are shown in Table 5. Interval labels can be used to decide the reachability relation on trees in $O(1)$ time, where node v is reachable from u iff $I_u \leq I_v$ and $E_v \leq E_u$, *i.e.*, $[I_v, E_v] \subseteq [I_u, E_u]$. After this step, we assign the interval labels to all cross edges to represent the ξ-reachable sub-trees induced by those cross edges, as illustrated in Figure 3. Specifically, given a cross edge $x \xrightarrow{\omega} y$, its sub-tree interval formed by the nodes that are ξ-reachable from x is $[I_y, E_z]$, where z is the target of tree edge $y \xrightarrow{\omega} z$. If the largest label of the tree edges of y is less than ω, y is the unique node that is ξ-reachable from x and the interval label for $x \xrightarrow{\omega} y$ is defined to be $[I_y, I_y]$. In addition, every *PES* is also encoded as the interval label of its origin node. For example, *PES* o_1 is encoded as $[0, 3]$.

Since a cross pair consists of two pointers that are ξ-reachable from the same origin node o, we directly combine the interval labels of two sub-tress and construct a *rectangle* label to encode the cross pairs obtained from the two sub-trees. Formally, a rectangle label is in the form $<X_1, X_2, Y_1, Y_2>$, where $[X_1, X_2]$ and $[Y_1, Y_2]$ are disjoint interval labels for two sub-trees. They are disjoint because the two sub-trees belong to different *PES*. Due to the disjointness, in a rectangle label, we always refer X_1 and X_2 to be the smaller timestamps, *i.e.*, $X_1 \leq X_2 < Y_1 \leq Y_2$. With rectangle labels, the **IsAlias(p, q)** query can be answered by testing whether or not the point (p, q) is enclosed by a rectangle in $O(\log n)$ time with an appropriate data structure.

Next, we present our efficient algorithm to generate and encode rectangle labels. We first classify the cross pairs. Suppose two pointers p and q belong to *PES* o_a and *PES* o_b, respectively. A cross pair belongs to two cases:

- *Case-1 pair.* p also points to o_b or q also points to o_a. For example, pointer p_7 in Figure 2 points to o_3, p_4 in *PES* o_1 also points to o_3. Therefore, p_4 and p_7 form a *Case-1* pair.
- *Case-2 pair.* Both pointers p and q point to a third object o_c, where $o_c \neq o_a$ and $o_c \neq o_b$. For example, pointers p_1 and p_7 in Figure 2 both point to o_5 but none of them belong to *PES* o_5.

Similarly, rectangles are also classified as *Case-1* and *Case-2* rectangles, which encode the *Case-1* and *Case-2* cross pairs respectively. Moreover, the *Case-1* rectangle $<X_1, X_2, Y_1, Y_2>$ also encodes the points-to information since Y_1 is the pre-order timestamp of an origin node. For instance, in rectangle $<1, 2, 5, 6>$, 5 is the timestamp for o_3.

The generation process of rectangle labels is illustrated in Table 6. We visit Pestrie in the object order to pair the cross edges belonging to each origin node. The importance of using the object order will be explained shortly. Consider the origin node o_3, which has a single cross edge $o_3 \rightarrow p_3$ that induces the sub-tree $\{p_3, p_4\}$, labeled as $[1, 2]$. Meanwhile, *PES* o_3 is encoded as $[5, 6]$. Of course, the nodes in the sub-trees $\{p_3, p_4\}$ and $\{o_3, p_7\}$ form *Case-1* cross pairs, hence we encode these pairs succinctly by a rectangle label: $<1, 2, 5, 6>$, a combination of the interval labels for corresponding sub-trees.

For *PES* o_5, the sub-tree labels for the cross edges $o_5 \rightarrow p_3$, $o_5 \rightarrow p_1$, and $o_5 \rightarrow p_7$ are $[1, 1]$, $[3, 3]$, and $[6, 6]$, respectively. The label for $o_5 \rightarrow p_3$ is $[1, 1]$ because the tree edge label $p_3 \rightarrow p_4$ is

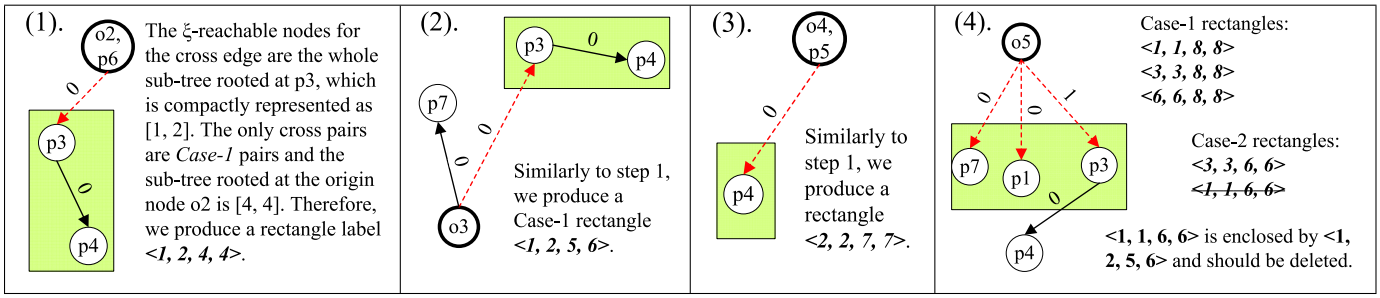

Table 6. Generating encoded cross pairs. We enumerate the origins in the *object order* to compute the cross pairs. The shadowed areas in the box are the ξ-reachable nodes for corresponding cross edges.

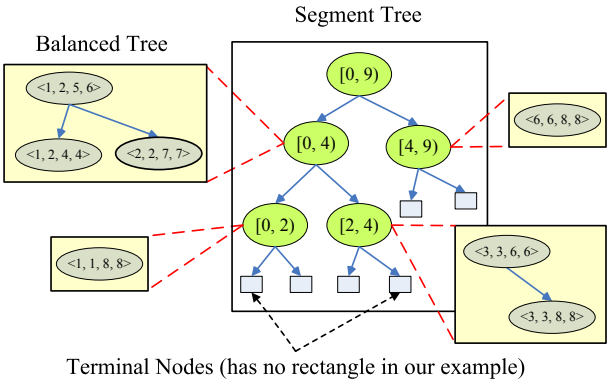

Figure 4. Rectangles generated from our sample points-to matrix.

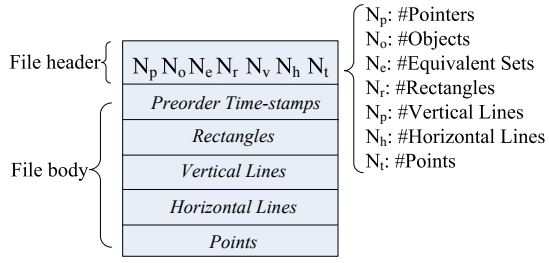

Figure 5. Layout of the encoding file.

0, which is smaller than the ξ-value 1 of $o_5 \rightarrow p_3$. Therefore, p_4 is not ξ-reachable from o_5 according to the ξ-condition. Note that the rectangle <1, 1, 6, 6> obtained by pairing $\{p_3\}$ with $\{p_7\}$ is enclosed by the rectangle <1, 2, 5, 6> that pairs $\{p_3, p_4\}$ with $\{o_3, p_7\}$. Therefore, the redundant rectangle <1, 1, 6, 6> needs to be discarded. We can find and delete all redundant rectangles according to the following theorem:

Theorem 2. *If we visit the PES origin nodes in the object order, a newly generated rectangle is either completely inside or outside of the previously generated rectangles.*

Theorem 2 indicates that a rectangle never overlaps with other rectangles. Therefore, if the corner point (X_1, Y_1) is covered by a rectangle $\mathcal{R}$, the whole rectangle $<X_1, X_2, Y_1, Y_2>$ must be covered by $\mathcal{R}$. We employ a *segment tree* to quickly retrieve the point cover information. The initial segment is $[0, N_e)$, where N_e is the number of Pestrie nodes. Every node of the segment tree represents a segment interval $[a, b]$, where the middle point is $mid = \frac{a+b}{2}$. Inside tree node, we use a balanced tree to store the rectangles that are intersected by the vertical line $x = mid$. All existing rectangles are stored in the balanced trees sorted by their Y_1 coordinates. When a new rectangle $\mathcal{R}$ is generated, we test whether its corner point (X_1, Y_1) is covered by existing rectangles. If it is covered, we discard $\mathcal{R}$ directly. The final structure for our example is shown in Figure 4.

Theoretically, encoding cross pairs takes $O(mn^2 \log^2 n)$ time. This is because, for each *PES*, we pair at most $O(n^2)$ sub-trees. Hence, in total, we pair $O(mn^2)$ sub-trees for m *PES*s. For each sub-tree pair, we execute the point enclosure query and it is performed in $O(\log^2 n)$, because we visit $O(\log n)$ segment tree nodes and spend $O(\log n)$ time for searching the balanced tree at every segment tree node. For each uncovered rectangle, inserting it into the segment tree takes $O(\log n)$ time. The size of encoded cross pairs is $O(R + n)$, where R is the number of stored rectangles and it is bounded by $O(n^2)$.

3.4.2 Generating Persistent File

We store the encoded rectangles in a file on disk. The format of the encoding file is depicted in Figure 5. The first row is the file header that specifies the dimensions of the Pestrie structure and the quantities of various types of rectangles. The second row contains the pre-order timestamps of all pointers and objects, which are the timestamps of their enclosing *ES* groups. The next four rows describe rectangles. We split the rectangles into four cases: points, vertical lines, horizontal lines, and rectangles. In this way, we reduce the encoding size because a substantial number of the rectangles are points and lines, which can be encoded by two and three integers, respectively. For example, five of the seven rectangles in Figure 4 are points and one of them is a line, which requires only thirteen integers to be stored in the persistent file.

4. Querying Encoded Information

We decode the persistent file and construct the querying structure in two steps. In the first step, we infer the *PES* identifiers for pointers. Those identifiers are essential to the **IsAlias** query but discarded in order to generate the small persistent file. To obtain the *PES* identifiers, we first load all pointers and objects and sort them using their pre-order timestamps. The sorted order is the object order used for Pestrie construction. Next, for the pointer p with the pre-order timestamp I_p, we use a binary search to determine the value for k such that $I_k \leq I_p < I_{k+1}$, where I_k and I_{k+1} are the pre-order timestamps for the k^{th} and $(k+1)^{th}$ objects in the object order. The value k is the *PES* identifier of p, because the pre-order timestamp of any node t with *PES* identifier k must have $I_t \in [I_k, I_{k+1})$.

In the second step, we load the rectangles and build a static query structure. Specifically, we maintain N_e lists and denote each list as $ptList[k]$, where N_e is the number of *ES* groups. For each

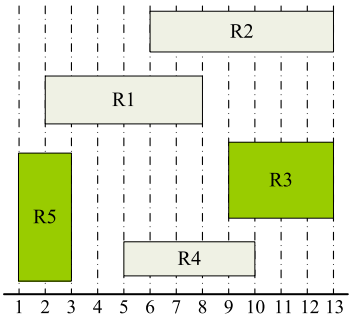

Figure 6. Query Structure. *ptList*[k] records the rectangles that intersect the vertical line $x = k$. R3 and R5 are *case-1* rectangles, where we can extract points-to information.

input rectangle $<X_1, X_2, Y_1, Y_2>$, we insert it into lists *ptList*[a], where $a \in [X_1, X_2]$. For example, the rectangle $R1$ in Figure 6 is inserted into lists *ptList*[2], *ptList*[3], ..., *ptList*[8]. We also generate a rectangle $<Y_1, Y_2, X_1, X_2>$ and insert it into lists *ptList*[b], where $b \in [Y_1, Y_2]$, because the alias relation is symmetric and we need the full alias relations to answer the **ListAliases** query. Finally, for each list *ptList*[i], $0 \le i < N_e$, we sort the referenced rectangles by their Y_1 coordinates.

Although a rectangle is referenced by multiple lists in our static query structure, the increase of memory consumption is not apparent since the numbers of horizontal lines and rectangles are much smaller than the numbers of vertical lines and points. Next, we use the query structure to answer queries.

IsAlias(p, q): We first compare the *PES* identifiers of p and q to test if they are an internal pair. If they are not, we use a binary search on *ptList*[I_p] to test if point (I_p, I_q) is covered by a rectangle, where I_p and I_q are the pre-order timestamps of p and q. This procedure works in $O(\log n)$.

ListAliases(p): Suppose the pre-order timestamp of p is I_p. The pointers that are aliased to p are encoded by the rectangles that intersect with the vertical line $x = I_p$. Therefore, we visit the rectangles in *ptList*[I_p]. For each rectangle $<X_1, X_2, Y_1, Y_2>$, the pointers with pre-order timestamps in the range $[Y_1, Y_2]$ are all aliased to p. The algorithm works in $O(K)$ time, where K is the size of answer set.

ListPointsTo(p): For each rectangle $\mathcal{R} = <X_1, X_2, Y_1, Y_2>$ in *ptList*[I_p], where I_p is the pre-order timestamp of p, we output the points-to relation $x \rightarrow Y_1$ if $\mathcal{R}$ is a *Case-1* rectangle. To further shorten the query time, we can recover the points-to matrix *PM* and directly return *PM*[p] as the answer.

5. Pestrie Optimization

We can tune the object order to achieve the minimal encoding size and shortest querying time, because different object orders produce different Pestrie. More specifically, our goal is to minimize the number of cross edges and to maximize the number of internal pairs. In this section, we study these optimization opportunities.

5.1 Theoretical Barrier

When using different object orders to construt Pestrie, the number of cross edges in the generated Pestrie are different, becuse the cross edges can be shared in different ways. In fact, we show that the sharing scheme of cross edges in Pestrie is very similar to the sharing scheme of nodes in standard Trie (Theorem 4 in Appendix A).[3] Therefore, minimizing the number of cross pairs can potentially reduce the encoding size, which is defined as:

[3] Therefore, we name our data structure Pestrie (*PES* trie).

Optimal Pestrie ***Construction Problem (OPC)***: *Finding an object order π to construct a* Pestrie *with the minimum number of cross edges.*

Since internal pairs can always be encoded in linear space and queried in $O(1)$ time, the second optimization problem is maximizing the number of internal pairs, which is defined as:

Optimal Pointer Partition Problem (OPP): *Given n pointers, m objects, a points-to matrix PM, and m groups $A_1, A_2, \ldots, A_m$. An ordering π is a permutation of the groups, where $\pi_i = j$ means group A_j is placed at the i^{th} position. With an ordering π, we put pointer p in group π_i iff $PM[p][\pi_i] = 1$, and $\forall k < i$, $PM[p][\pi_k] = 0$. The* OPP *problem asks for an order π to maximize the function O_π defined as follows:*

$$Maximize: \quad O_\pi = \sum_{i=1}^{m} I_i^2, \quad I_i = \left|A_{\pi_i}\right|$$

Unfortunately, both the OPC and OPP problems are NP-hard and the proofs can be found in Appendix A. Therefore, we in turn search for a good heuristic.

5.2 Heuristic

We use the object order obtained by sorting their hub degrees to construct Pestrie. The object order essentially makes pointer partitions uneven. We show in Theorem 3 that the optimal solution of OPP problem only depends on σ^2, where σ is the standard deviation of I_i. It is maximized if the distribution of pointers is uneven.

Theorem 3. *For any ordering π, $O_\pi = m\sigma^2 + \frac{n^2}{m}$.*

Proof. First, note that:

1. $\sum_{i=1}^{m} I_i = n$;
2. Let $\bar{a} = \frac{n}{m}$, we have $m\bar{a}^2 = \bar{a}n = \frac{n^2}{m}$.

For any permutation π, we have the transformation:

$$\begin{aligned}
\frac{O_\pi}{m} &= \frac{\sum_{i=1}^{m} I_i^2}{m} \\
&= \frac{(\sum_{i=1}^{m} I_i^2) - 2\bar{a}n + m\bar{a}^2}{m} + \bar{a}^2 \\
&= \frac{\sum_{i=1}^{m} (I_i^2 - 2\bar{a}I_i + \bar{a}^2)}{m} + \bar{a}^2 \\
&= \frac{\sum_{i=1}^{m} (I_i - \bar{a})^2}{m} + \bar{a}^2 \\
&= \sigma^2 + \bar{a}^2
\end{aligned}$$

$$\Longrightarrow O_\pi = m\sigma^2 + \frac{n^2}{m}$$ □

The OPC problem also favors our heuristic. As shown by Comer, the greedy heuristic that selecting an attribute at each level which adds the smallest number of nodes to the next level almost builds an optimal Trie [6]. Since Pestrie is a variant of Trie (Section A.2), Comer's heuristic implies the guideline that pointers with similar points-to sets should be kept in the same ES node as long as possible during Pestrie construction. Since the pointers with large points-to sets (we refer to them as L-pointers) could derive more cross edges, L-pointers should be picked in each step of the partitioning as much as possible according to Comer's heuristic. Since the L-pointers likely point to common objects, especially those that have large hub degrees, first using the objects with larger hub degrees to partition pointers can potentially keep these pointers staying in the same ES node for longer time.

6. Implementation

6.1 Preparing Points-to Matrix

Our boolean matrix representation for pointer information is not the default format for all points-to algorithms. Therefore, we need to canonicalize the input information. Our matrix representation (*PM*), in essence, is the standard representation for the flow- and context-insensitive points-to results, which does not subsume the *constrained* points-to information produced by various pointer analysis. For example, the flow-sensitive points-to information, such as p points to o at the program point l, is represented as $p \xrightarrow{l} o$. Fortunately, the constrained points-to representation can be transformed to our binary matrix easily.

In the flow-sensitive analysis, such as the one developed by Lhoták *et al.* [19], a points-to relation $p \xrightarrow{l} o$ is first represented as $(l, p) \rightarrow o$, where (l, p) describes the version of pointer p defined at program point l. We can map every location pointer pair (l, p) to a new pointer p^l, which is mapped to a unique integer that represents a row in *PM*. Therefore, the original points-to relation is encoded in a unconstrained format $p^l \rightarrow o$.

Transforming the context-sensitive information is similar to transforming flow-sensitive information. A context-sensitive points-to relation can be encoded as $p \xrightarrow{c} o$, where the constraint c is the context condition for this points-to relation. Again, we first rewrite the expression $p \xrightarrow{c} o$ as $(c, p) \rightarrow o$. Then, we replace all occurrences of (c, p) with a new pointer p^c and construct the points-to relation $p^c \rightarrow o$. If the object is also constrained with a context, *e.g.* $(c_1, p) \rightarrow (c_2, o)$, we also replace (c_2, o) with o^{c_2}.

The result of the full context-sensitive analysis, such as the one developed by Xiao *et al.* [40], can also be transformed to our format. However, we cannot fully represent the points-to relations with all contexts, which could exceed 2^{64}. For our experiments, we first generate the *1*-callsite-sensitive result by merging the contexts. We then transform the *1*-callsite-sensitive information to our format. We exemplify how to generate *1*-callsite-sensitive information for the result produced by geomPTA [40], which is used in our evaluation. In the first step, we merge all contexts $c_1, c_2, \cdots, c_k$ that are introduced by the same callsite into a single *representative context* C. Then, for every points-to relation $(c_1, p) \rightarrow (c_2, o)$, we replace it with $(C^1, p) \rightarrow (C^2, o)$, where C^1 and C^2 are the representative contexts for c_1 and c_2, respectively.

Path-sensitive points-to information, such as the work by Hacket *et al.* [12], can also be transformed to binary matrix format. The basic idea is finding a finite set of basis logic expressions and rewriting every path condition as a disjunction of these expressions. This is similar to representing every vector with a linear combination of basis vectors. In this way, a points-to relation $p \xrightarrow{l} o$ is written as $p \xrightarrow{l_1 \vee l_2 \vee \cdots \vee l_k} o$, where l_1, l_2, $\cdots$, l_k are chosen from the basis logic predicates set. The points-to relation can be split into multiple relations $p \xrightarrow{l_1} o \cup p \xrightarrow{l_2} o \cup \cdots \cup p \xrightarrow{l_k} o$. Then, every pair of pointer and basis logic predicate (p, l_1) can be mapped to a new pointer p^{l_1}. Finally, $p \xrightarrow{l} o$ is transformed to a vector of $p^{l_1} \rightarrow o$, $p^{l_2} \rightarrow o$, $\cdots$, $p^{l_k} \rightarrow o$.

6.2 Variable Correlation across Analysis Cycles

Since all variables in persistent files are mapped to integers, we should keep the mapping consistent to incorporate the persistent pointer information used in program analysis tools. To achieve the goal, we save to disk the IR produced by the program analysis tool and the mapping from variable names to integers produced in the first run. In future runs, we directly load the saved contents to avoid rebuilding the IR and guarantee original variable mapping. In addition, the call graph and mapping from call edges to integers are also saved for future use. After the transformation of the context-sensitive points-to result to binary matrix, we map each context pointer pair (c, p) to p^c, where c consists of the integers for corresponding call edges. To answer queries such as **ListPointsTo(c, p)**, the naming of the call edges should also be consistent across analysis runs in order to find the correct p^c that (c, p) is mapped to.

7. Experiments

In this section, we evaluate the compression capability and the querying efficiency of the Pestrie persistent scheme, by comparing to the bitmap-based encoding scheme, the BDD encoding, and the off-the-shelf bzip compressor. The subjects in Table 2 are also used in our experiments. Both the bitmap and the Pestrie algorithms are written in C++, compiled by g++ 4.8 with -O2 option.[4] The sparse bitmap implementation is taken from the GCC compiler, which is a highly optimized library. We use the default 128 bits for each sparse bitmap block, which is optimal in our evaluation. We conduct the experiments on a 64bit machine with an Xeon 3.0GHz processor and 32GB main memory, running Ubuntu 13.04.

7.1 Performance Evaluation

We first generate the encoded pointer information. Our Pestrie approach generates a single file Pes_P as shown in Figure 5. The bitmap approach generates a file Bit_P that encodes the points-to matrix *PM* and the alias matrix *AM*. The BDD and bzip approaches encode only the points-to matrix *PM*, referred to as BDD and bzip, respectively. Next, we use a real client to evaluate their querying performance.

7.1.1 Querying Performance

We assess the querying efficiency by computing aliasing pairs [25] — data conflicting load and store statements pairs — in two ways. The first method extracts the base pointers for all the stores and loads.[5] Then, it enumerates all pairs of base pointers and uses the **IsAlias** query to determine if they have an access conflict. The second method uses the **ListAliases(p)** query to retrieve the list of base pointers that are aliased to a given base pointer p. The querying performance results are collected in Table 7.

Querying Time. We observe that, for **IsAlias** query, the most popular query for the alias information, using Pes_P is on average (by geometric mean) 1.6× faster than using Bit_P. Pes_P is faster because locating a bit in sparse bitmap is not in constant time. In fact, in GCC's implementation, it uses a linked list to manage sparse bitmap blocks, which needs $O(n)$ time to scan the linked list to determine the existence of a bit. In Pestrie, the search only takes $O(\log n)$ time. For **ListAliases** query, both Pes_P and Bit_P are efficient, because the querying results have been pre-computed and the query can be answered by only outputting the pre-computed results.

We also evaluate the on-demand versions of these queries. The results are collected in Table 7 with "Demand" label. To mimic the conventional usage, we only use the *PM* matrix to evaluate queries. Specifically, **IsAlias(p, q)** is answered by *intersecting the points-to sets of p and q*, which takes 2.9× more time than querying with Pestrie. In case of **ListAliases(p)** query, we execute **IsAlias(p, q)** with all other base pointers q and cache the querying result in *cache(p)*. Next time we query **ListAliases(p')**, where p' is an equivalent pointer to p, we directly use the cached result *cache(p)* as the answer. With the cache optimization, the demand version of **ListAliases** is 2× faster than that of **IsAlias**, though it is still 123.6× slower than the Pestrie-based **ListAliases**. Since the demand-driven **IsAlias** is the unique interface for many tools,

[4] Code can be found at: `https://github.com/richardxx/pestrie`.

[5] For example, in a store $p.f = q$, p is the base pointer.

Program	IsAlias (s)			ListAliases (s)			ListPointsTo (s)		Decoding Time (s)		Querying Memory (MB)	
	Pes_P	Bit_P	Demand	Pes_P	Bit_P	Demand	Pes_P	BDD	Pes_P	Bit_P	Pes_P	Bit_P
samba	62.7	66.2	103.7	0.02	0.04	55.3	0.01	-	0.12	0.13	47.7	144.0
gs	36.5	45.8	146.6	0.23	0.64	81.1	0.11	-	0.19	0.22	37.2	157.8
php	65.2	120.9	745.1	1.3	2.1	350.5	0.4	-	0.31	0.31	52.7	215.7
postgreSQL	51.4	101.6	843.2	2.8	4.1	365.3	0.91	-	0.44	0.38	53.4	249.5
antlr	21.6	28.3	35.1	0.07	0.28	26.7	0.03	43.2	0.2	0.16	45.6	81.9
luindex	17.5	23.7	28.7	0.04	0.11	22.0	0.02	30.2	0.2	0.13	42.2	98.4
bloat	78.3	101.2	134.2	0.18	0.14	105	0.09	138.3	0.67	0.28	92.0	210.0
chart	107.2	124.2	207.2	0.28	0.2	147.9	0.3	602.6	0.99	0.4	154.9	446.5
batik	48	114.5	117.6	4.0	3.9	30.3	0.18	-	11.9	2.18	782.2	939.1
sunflow	26.7	56.4	68.5	1.9	2.2	26.5	0.12	-	7.3	1.4	555.2	623.1
tomcat	32.6	79.6	71.3	3.4	3.5	29.6	0.15	-	15.7	2.6	1186.1	917.8
fop	105.1	209.9	205.9	9.2	7.8	57.5	0.31	-	16.7	3.3	1165.8	1350.1

Table 7. Summary of query time, persistence loading time, and querying structure memory size.

such as Soot and LLVM, we argue that a well designed persistence scheme and a comprehensive query interface are very helpful for speeding up client applications.

An interesting observation is that the Bit_P version of **IsAlias** is not always faster than the demand version, such as in tomcat and fop. The reason is that both versions of **IsAlias** take $O(n)$ time and their performance is significantly sensitive to the input. On the other hand, the $O(\log n)$ complexity of Pes_P is a strong guarantee for time critical applications.

We also compare the performance of answering the **ListPointsTo** query with Pes_P and BDD. The pointers used for query evaluation are also the base pointers. The BDD-based queries are directly answered through the Paddle interface with the JavaBDD backend [20]. Comparatively, BDD is 1609.6× slower. Although Paddle is implemented in Java, the significant performance gap cannot be primarily contributed by the language difference. Therefore, BDD is not a compelling choice for query-intensive applications.

Persistence Decoding Time. We measure the time to decode the persistent information for querying. As shown in Table 7, both Pestrie and bitmap encodings can be decoded efficiently. The largest case fop only takes 16.7s, while the points-to analysis for fop consumes 1.7 hours. Even more, the points-to analysis on chart takes 4.6 hours, but decoding its Pestrie persistence file only takes 0.5s. Therefore, with persistent pointer information, clients can start their own jobs more efficiently.

Query Memory Usage. Query memory is the memory consumed by the querying structure. As shown in Table 7, the largest program fop only uses 1165MB of memory to maintain the query structure, whereas the pointer analysis for fop uses 22GB memory. Another observation is that Pes_P consumes less memory than Bit_P in all cases except tomcat. This proves that the static querying structure of Pes_P does not increase memory usage significantly.

7.1.2 Persistence Generation Performance

Storage Size. We summarize the sizes of persistent files for all four encoding approaches in Table 8. On average, the size of Pes_P is 10.5× smaller than the size of Bit_P, which proves that the hub property is a good observation for compressing alias matrix. Moreover, Pes_P is 17.5× smaller than BDD and 39.3× smaller than bzip. Note that the size of BDD and bzip encoding in our experiments is only the size of the points-to matrix *PM*, which is much smaller than the alias matrix *AM*. However, with *PM* alone, we can only answer aliasing queries on demand.

Persistence Construction Time. As shown in Table 8, constructing the bitmap persistent encoding is faster for the first eight programs, and constructing the Pestrie encoding is faster for the last four programs. The sparse bitmap is efficient for calculating $PM \times PM^T$ if the matrix is *sparse*. Specifically, for pointer p, the

Program	Storage (MB)				Construction Time (s)		
	Pes_P	Bit_P	BDD	bzip	Pes_P	Bit_P	bzip
samba	5.3	20.4	-	3.2	3.5	0.7	120.3
gs	3.6	30.1	-	45	19.9	5.3	690.2
php	3.5	46.7	-	141	38.7	12.6	2259.9
postgreSQL	3.1	54.5	-	572	65.1	16.9	7023.1
antlr	2.6	13.0	45.0	9.2	1.2	0.8	75.7
luindex	2.3	12.7	40.0	7.9	1.1	0.7	59.4
bloat	5.5	46.6	92.1	33	7.5	4.4	284.2
chart	8.6	58.3	158.5	380	12.1	6.5	1628.2
batik	7.0	172.7	-	5300	109.0	344.4	16106.4
sunflow	4.7	113.4	-	2200	62.0	228.7	8349.5
tomcat	5.0	146.3	-	1900	57.4	545.0	8765.8
fop	26.9	255.7	-	15000	169.9	615.4	21183.1

Table 8. Summary of encoding size and construction time.

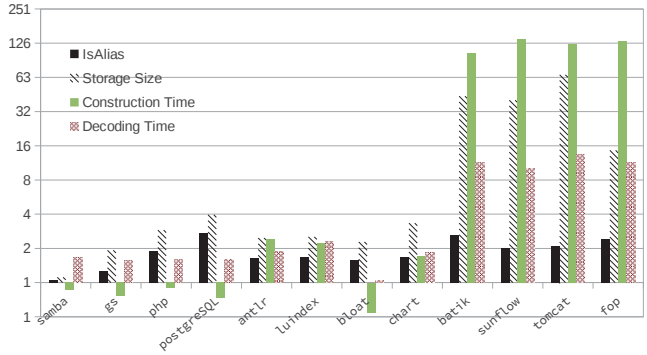

Figure 7. Impact of object order on Pestrie performance.

alias set of p is the union of the rows o in PM^T, where p points to o. Merging is fast when *PM* is sparse since only a few pointers are visited. However, for the **IsAlias** query, we intersect the points-to set of p with the points-to sets of *all* other base pointers. Most of them are not aliased with p. This is why the demand version of **IsAlias** even takes more time than constructing the bitmap encoding in the first eight cases. When the points-to matrix becomes dense, the merging process wastes a significant amount of time to merge the same pointers multiple times. However, the dense matrix is favored by Pestrie, because large rectangles can be generated and many redundant rectangles are pruned by Theorem 2. This is why Pestrie is faster for the last four subjects.

7.2 Heuristic Effectiveness

The object order obtained by sorting hub degrees is effective for constructing compact and query-efficient Pestrie persistence. To prove our claim, we compare Pes_P to Pes_{rand}, a Pestrie persistence constructed by random object order. The comparison result is summarized in Figure 7, which plots the ratio of the time or the space

consumption of Pes_{rand} over Pes_{P}. On average, decoding Pes_{rand} takes 3.2× more time and answering **IsAlias** query is 1.8× slower, compared to Pes_{P}. Generally, Pes_{rand} has many more cross edges, which result in a large number of small rectangles. Also, due to the additional cross edges, it takes 5.3× more time to generate the Pes_{rand} encoding. An interesting observation is that Pes_{rand} generation is faster in some cases such as gs, because the cross edges in Pes_{rand} are more evenly distributed and thus, less point enclosure queries are issued. Finally, the persistent file produced by Pes_{rand} is 5.9× larger, which is also caused by the additional cross edges.

8. Related Work

Modular Points-to Analysis. There is a large body of work on building *function summaries* for incremental and scalable program analysis [9, 12, 45]. However, function summaries cannot be queried *directly*, as they should be linked to build the whole program information before answering queries. Function summaries are usually small, hence they do not need special treatment for compaction. For example, on average of 0.075 entry aliasing edges and 0.391 exit aliasing edges are generated by Hackett *et al.* [12], which are summed up to only several megabytes even for a large program. In contrast, the whole program information always contains gigabytes of data as shown in our experiment. Moreover, linking summaries to build the whole program information takes a long time so that using summaries cannot quickly boot the applications based on pointer information.

Demand-driven Points-to Analysis. Unlike modular analysis, demand-driven points-to analysis can provide short time and small memory footprints for querying pointer information [34, 42, 44, 48]. However, the demand-driven approach cannot be used in query-intensive situation due to its long query processing time [42]. Moreover, it is unknown how to answer the **ListAliases** query efficiently in a demand-driven manner. Therefore, persistent pointer information is more attractive for query-intensive applications.

Encoding Pointer Information. Whaley *et al.* are the first to store points-to relations in BDDs to support higher order context-sensitive analysis compactly [17, 39]. Due to the availability of precise points-to information, Martin *et al.* develop a defects analysis engine [23] and Naik *et al.* implement Chord, a system for static race and deadlock detection [25, 26]. All these systems successfully demonstrate the importance of encoding points-to information to save recurring time for points-to analysis. However, as we showed, BDDs are not query efficient. Our encoding scheme, Pestrie, can be more compact and query-efficient than BDDs.

Other than BDD, Le *et al.* describe a bitmap encoding for Mod-Ref information [18], which helps a JIT compiler for aggressive online optimizations. Their approach for compacting Mod-Ref information is similar to our bitmap encoding for pointer information described in Section 2.1, which is shown less effective compared to the Pestrie approach.

9. Conclusion and Future Work

In this paper, we present a compact and query-efficient persistence scheme Pestrie for pointer information. Pestrie serves queries efficiently and achieves small persistence storage size. The main focus of our future work is applying persistence technique to precompute pointer information for libraries in order to reduce the cost of points-to analysis for framework-heavy programs.

Acknowledgments

We especially thank Atanas Rountev, who proofread this paper and proposed numerous improvement suggestions. We also thank the PLDI reviewers for their useful feedback. This research is supported by RGC GRF grant RGC622909 and RGC621912 and also by the HKUST RFID Center.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, 1989.

[2] K. Ali and O. Lhoták. Averroes: Whole-program analysis without the whole program. In *ECOOP*, 2013.

[3] E. Bodden. Pointer analyses for open programs. In O. Lhotak, Y. Smaragdakis, and M. Sridharan, editors, *Pointer Analysis (Dagstuhl Seminar 13162)*. 2013.

[4] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

[5] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, 2009.

[6] D. Comer. Analysis of a heuristic for full trie minimization. *ACM Trans. Database Syst.*, Sept. 1981.

[7] D. Comer and R. Sethi. The complexity of trie index construction. *J. ACM*, July 1977.

[8] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS*, 2001.

[9] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, 2011.

[10] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA*, 2004.

[11] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, and D. Weiss. Customization change impact analysis for ERP professionals via program slicing. In *ISSTA*, 2008.

[12] B. Hackett and A. Aiken. How is aliasing used in systems software? In *FSE*, 2006.

[13] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, 2007.

[14] B. Hardekopf and C. Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *SAS*, 2007.

[15] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, 2011.

[16] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 1999.

[17] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *PODS*, 2005.

[18] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in JIT optimizations. In *CC*, 2005.

[19] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *POPL*, 2011.

[20] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 2008.

[21] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE*, 2011.

[22] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. In *FSE*, 2004.

[23] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, 2005.

[24] N. A. Naeem and O. Lhotak. Typestate-like analysis of multiple interacting objects. In *OOPSLA*, 2008.

[25] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.

[26] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, 2009.

[27] C. M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[28] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *PLDI*, 2000.

[29] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC*, 2001.

[30] A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *FSE*, 1999.

[31] A. Rountev, M. Sharp, and G. Xu. IDE dataflow analysis in the presence of large object-oriented libraries. In *CC*, 2008.

[32] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, 2011.

[33] M. Sridharan. Practical aspects of pointer analysis. In O. Lhotak, Y. Smaragdakis, and M. Sridharan, editors, *Pointer Analysis (Dagstuhl Seminar 13162)*. 2013.

[34] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, 2006.

[35] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.

[36] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA*, 2012.

[37] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, 2009.

[38] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC*, 2000.

[39] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[40] X. Xiao and C. Zhang. Geometric encoding: forging the high performance context sensitive points-to analysis for Java. In *ISSTA*, 2011.

[41] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA*, 2008.

[42] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, 2009.

[43] G. Xu, D. Yan, and A. Rountev. Static detection of loop-invariant data structures. In *ECOOP*, 2012.

[44] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*, 2011.

[45] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *POPL*, 2008.

[46] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO*, 2010.

[47] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *PLDI*, 2013.

[48] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL*, 2008.

A. Theorems

A.1 Proof for Theorem 2

Suppose there are two cross edges, e_1: $r_1 \xrightarrow{\xi_1} y_1$ and e_2: $r_2 \xrightarrow{\xi_2} y_2$, where y_1 and y_2 belong to the same *PES*. Assume e_1 is created earlier than e_2. We have the following lemma.

Lemma 2. *If* $[I_{e_1}, E_{e_1}] \cap [I_{e_2}, E_{e_2}] \neq \emptyset$, *we have* $[I_{e_2}, E_{e_2}] \subseteq [I_{e_1}, E_{e_1}]$.

Proof.
- $y_1 = y_2$: This implies $\xi_1 < \xi_2$. Clearly, $I_{e_1} = I_{e_2}$ and $E_{e_1} > E_{e_2}$. The result is immediate.
- $y_1 \neq y_2$: y_1 must be an ancestor of y_2. Let $S = [I_{e_1}, E_{e_1}] \cap [I_{e_2}, E_{e_2}]$. Clearly, $\forall y' \in S$, y' is ξ-reachable from e_1 and is ξ-reachable from e_2, thus, the path $y_1 \rightsquigarrow y'$ must pass y_2. Therefore, y_2 is also ξ-reachable from e_1 and, in turn, all nodes in the sub-tree of y_2 are ξ-reachable from e_1, *i.e.* $[I_{e_2}, E_{e_2}] \subseteq [I_{e_1}, E_{e_1}]$.

□

Theorem 2. *If we visit the PES origin nodes in the object order, a newly generated rectangle is either completely inside or outside of the previously generated rectangles.*

Proof. Suppose we pair two cross edges e_1: $r \xrightarrow{\xi_1} y_1$ and e_2: $r \xrightarrow{\xi_2} y_2$ to generate a rectangle label R_1: $< I_{e_1}, E_{e_1}, I_{e_2}, E_{e_2} >$, where (I_{e_1}, I_{e_2}) is the lower left point. Suppose there is an existing rectangle R_2: $< I_{e_3}, E_{e_3}, I_{e_4}, E_{e_4} >$ generated by the cross edges e_3: $r' \xrightarrow{\xi_3} y_3$ and e_4: $r' \xrightarrow{\xi_4} y_4$. We have:

- (I_{e_1}, I_{e_2}) is enclosed by R_2, *i.e.* $[I_{e_1}, E_{e_1}] \cap [I_{e_3}, E_{e_3}] \neq \emptyset$ and $[I_{e_2}, E_{e_2}] \cap [I_{e_4}, E_{e_4}] \neq \emptyset$. According to Lemma 2, we have $[I_{e_1}, E_{e_1}] \subseteq [I_{e_3}, E_{e_3}]$ and $[I_{e_2}, E_{e_2}] \subseteq [I_{e_4}, E_{e_4}]$, *i.e.* R_1 is inside of R_2.
- (I_{e_1}, I_{e_2}) is not enclosed by R_2. If R_1 overlaps R_2, we must have $[I_{e_1}, E_{e_1}] \cap [I_{e_3}, E_{e_3}] \neq \emptyset$ and $[I_{e_2}, E_{e_2}] \cap [I_{e_4}, E_{e_4}] \neq \emptyset$. According to Lemma 2, R_1 is enclosed by R_2, which contradicts our assumption. Therefore, R_1 is outside of R_2.

□

A.2 The OPC Problem

We prove that the OPC problem is NP-hard. The main idea is establishing the relationship between the number of **cross edges** in the Pestrie G_{pes} (we refer to them $|G_{pes}|$) and the number of **nodes** in the standard Trie T_{std} (we refer to them $|T_{std}|$), where both G_{pes} and T_{std} are constructed by the same pointed-by matrix PM^T. In Trie terminology, a row in PM^T is a record and every object is an attribute [7]. The attribute testing order is exactly the object order for Pestrie. Of course, we use the same order for attribute testing and partitioning pointers. Specifically, T_{std} is built as follows:

Step 1. We build the root node V_{root} for T_{std}, initialize $tail_p = V_{root}$ for all pointers p and $tail_o = V_{root}$ for all objects o.

Step 2. We scan every row of PM^T and update T_{std} and the values for *tail*. Suppose we are scanning the i^{th} row and the corresponding attribute (object) is o_i. For each pointer p in the row, we build an edge $N_{old} \xrightarrow{o_i} N_{new}$ if it does not exist, where $N_{old} = tail_p$. Then, we update $tail_p$ to be N_{new}. After processing all the pointers in the i^{th} row, we process o_i in the same manner as a pointer. Therefore, in some abuse of terminology, we use object o_i and pointer o_i interchangeably.

Figure 8 demonstrates the insertion of the first four rows of the pointed-by matrix in Table 3 into a standard Trie. We have the following observation for the Trie construction:

Lemma 3. *After processing the* j^{th} *row of* PM^T, *we have* $|G_{pes}| = |T_{std}| - j$, $\forall 0 < j \leq m$, *where* m *is the number of rows in* PM^T.

Proof. We first collect o_j and all the pointers appearing in row j of PM^T in a container Φ. Then, we cluster the pointers in Φ into N_C classes by their *tail* values. Therefore, we will create N_C edges in T_{std} for row j (please review **Step 2**). Comparatively, in the Pestrie counterpart, before processing row j, the pointers are also residing in N_C different *ES* groups. For pointers p and q in the same class, we must have p and q have the same points-to sets before processing row o_j, otherwise $tail_p \neq tail_q$ and, thus, p and q are not in the same class. Therefore, the pointers in row j also belong to N_c different nodes in the Pestrie counterpart, and p and q belong to the same node in Pestrie. Since object o_j and the pointers not appeared in row i, where $i < j$, all have $tail = V_{root}$, we do not build cross edges for o_j and those pointers in Pestrie. Therefore, only

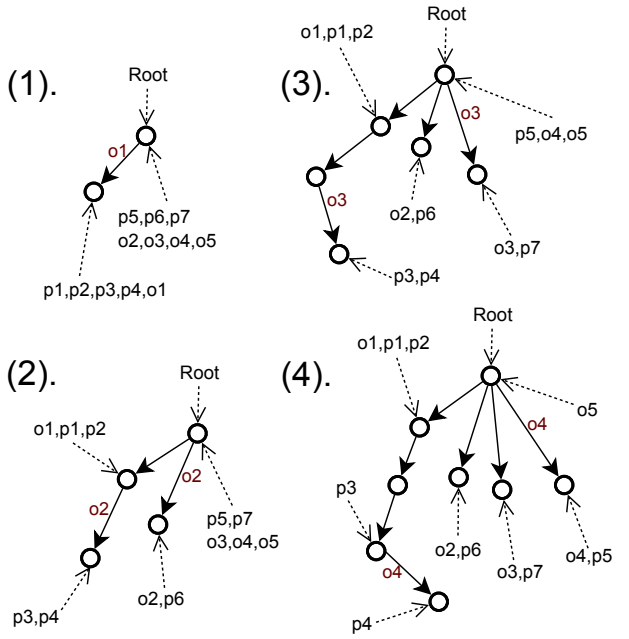

Figure 8. First four steps of constructing a standard Trie with the pointed-by matrix in Table 3. A solid arrow is the Trie edge and a dotted arrow is the $tail_p$ value of pointer p.

$N_c - 1$ cross edges are created in step j of Pestrie construction. As a consequence, after processing the j^{th} row, $|G_{pes}| = |T_{std}| - j$. □

Theorem 4. *The OPC problem is NP-hard.*

Proof. First, $|G_{pes}| = |T_{std}| - m$ after all m rows are processed, according to Lemma 3. Since the term m is irrelevant to object order chosen for constructing G_{pes}, minimizing $|G_{pes}|$ is equal to minimizing $|T_{std}|$. However, the optimal Trie problem is NP-hard [7], thus the OPC problem is NP-hard. □

A.3 The OPP Problem

We first define a new problem called MSS problem as follows:

Maximum Squared Sum Problem (MSS): *Given n elements and m groups, a binary matrix B where $B[i][j] = 1$ means that element i can be put into group j. We compute an arrangement M for the elements to maximize the objective function O_M, where $M[i] = j$ means that element i is assigned to group j:*

$$Maximize: \ O_M = \sum_{j=1}^{m} S_j^2, \ \text{Where:} \ S_j = |\{i | M[i] = j\}|$$

The difference between OPP problem (Section 5) and MSS problem is that, in OPP, each pointer p is kept in the first group A_i in the permutation π, where $PM[p][i] = 1$. For example, if $PM[p][a] = 1$ and $PM[p][b] = 1$, p will be assigned to group a if a ordered prior to b in the permutation π. However, in MSS problem, it is free to put an element e in any group g where $B[e][g] = 1$. The two problems are connected with the following lemma:

Lemma 4. *The* MSS *problem can be polynomially reduced to* OPP *problem, i.e.* MSS $\leq_p$ OPP.

Proof. We build a *configuration graph* G_M to describe an arrangement M in MSS, where each node represents a group. If we can transfer more than zero elements from group x to group y, we add an edge $x \xrightarrow{\delta} y$, where the label δ indicates the number of transferable elements from x to y. Moreover, S_x denotes the number of elements assigned to group x.

We first show that, if M is optimal, G_M is a DAG. Suppose there is an edge $x \xrightarrow{\delta} y$ where $S_x \leq S_y$. We can obtain a better solution if we transfer all δ elements from x to y. After we exchange δ elements, we have:

$$(S_x - \delta)^2 + (S_y + \delta)^2 = S_x^2 + S_y^2 + 2\delta \times (\delta + S_y - S_x)$$

Since $2\delta \times (\delta + S_y - S_x)$ is greater than zero ($\delta > 0$), the arrangement after the transfer is better than before. Hence, the optimal arrangement must be that, for any edge $x \xrightarrow{\delta} y$, we have $S_x > S_y$. Therefore, G_M is a DAG because, a cycle must include an edge $x \xrightarrow{\delta} y$ where $S_x \leq S_y$, which is a contradiction.

Second, suppose the groups are ordered. We show that an arrangement M has a DAG configuration graph G_M if, for any element a, a is assigned to the group g_0, where g_0 is the first group in the order and a can be assigned to groups $g_0, g_1, \ldots, g_k$. This is because, assigning elements in this way can only produce edges in the form $g_0 \rightarrow g_1, \cdots, g_0 \rightarrow g_k$. Therefore, we cannot have an edge $g_i \rightarrow g_j$, where j is ordered prior to i, *i.e.* G_M is a DAG.

Combing the two observations, by enumerating all ordering of the groups and for every order, assigning the elements x to the first group in the order that is permitted to hold x, we can obtain the optimal arrangement. This description is in fact equal to the OPP problem. Therefore, an instance of MSS problem is also an instance of OPP problem, *i.e.* MSS $\leq_p$ OPP.

□

Lemma 5. *The* MSS *problem is NP-hard.*

Proof. We reduce the exact cover by 3-sets problem (EX_3) to MSS problem. The EX_3 problem is defined as follows:

*EX_3: **Exact Cover by 3-sets**: Given a complete set S, and a list of subsets $s_1, s_2, \ldots, s_m$ carrying the elements in S. Each set s_i has exactly three elements and n, the number of elements in S, is divisible by three. The problem asks if we can select some subsets from the list, where their union is S and they are mutually disjoint.*

The NP-hardness proof of EX_3 can be found in [27]. Next, we reduce EX_3 to MSS. We treat every subset in EX_3 as a group in MSS, and $\forall a \in s_i$, we fill $B[a][s_i] = 1$. We claim that the answer of EX_3 is *Yes iff* the optimal answer of MSS is $3n$.

if: Any solution of the EX_3 problem must have $\frac{n}{3}$ sets. Corresponding to this solution, the answer for the reduced MSS problem is $3^2 \times \frac{n}{3} = 3n$.

only if: If the objective function O_M for an arrangement M is $3n$, every group in M has either 0 or 3 elements. Otherwise, suppose we have n_3, n_2, and n_1 groups that have 3, 2, and 1 elements, respectively. Clearly, $n_3 = \frac{n}{3}$ and $n_2 = n_1 = 0$ is a solution to the equation $O_M = 9n_3 + 4n_2 + n_1 = 3n$. Suppose we split a groups with 3 elements into b groups with 2 elements and c groups with 1 element. We have $n_3 = \frac{n}{3} - a$ and $n_1 + n_2 = a$. However, in such case, $9n_3 + 4b + c < 3n$ because, the quadratic function $f(x) = x^2$ has the following property:

$\forall a, b, c > 0$ *where* $a \times 3 = b \times 2 + c \times 1$,
we have $a \times f(3) > b \times f(2) + c \times f(1)$.

Therefore, after the splitting, the value of O_M becomes smaller. Thus, the only solution for $O_M = 3n$ is $n_3 = \frac{n}{3}$ and $n_2 = n_1 = 0$, which is a solution to the original EX_3 problem, *i.e.* $EX_3 \leq_P$ MSS and MSS is NP-hard. □

Theorem 5. *The* OPP *problem is NP-hard.*

Proof. Combining Lemma 4 and Lemma 5, the result is immediate. □