SIMD-Accelerated Sparse Bit-Vectors for Pointer Analysis

Zhaoyang Tan
Zhejiang University
The State Key Laboratory of Blockchain and Data Security
Hangzhou, China
tzyang@zju.edu.cn

Peisen Yao✉
Zhejiang University
The State Key Laboratory of Blockchain and Data Security
Hangzhou, China
pyaoaa@zju.edu.cn

Kui Ren
Zhejiang University
The State Key Laboratory of Blockchain and Data Security
Hangzhou, China
kuiren@zju.edu.cn

Abstract

Pointer analysis is a core component of program-analysis frameworks. Inclusion-based solvers repeatedly update points-to sets, and the resulting set operations frequently dominate end-to-end runtime. In existing sparse bit-vector representations, we find that the main cost is not bitmap arithmetic but irregular index matching: identifying allocation-site representatives shared by two operands before combining their corresponding blocks. We accelerate this kernel by recasting index matching as a branch-free SIMD primitive. The key is a sparse layout that stores allocation-site indices and their blocks in contiguous arrays, enabling parallel scans over indices and masked SIMD updates of blocks while preserving sparsity. We evaluate the technique on 12 real-world programs under both context-insensitive and context-sensitive variants. Compared to a conventional sparse bit-vector baseline, our approach reduces analysis time by up to 3.8× (geomean: 2.0×) and peak memory by up to 86% (geomean: 58%), with larger gains for more precise analyses. The improvements persist across alternative set representations and constraint solving strategies.

CCS Concepts

• **Theory of computation → Program analysis**.

Keywords

Pointer Analysis, Data Structures, SIMD Optimization

ACM Reference Format:
Zhaoyang Tan, Peisen Yao, and Kui Ren. 2026. SIMD-Accelerated Sparse Bit-Vectors for Pointer Analysis. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837421

1 Introduction

Pointer analysis [15, 16] is a foundational static analysis that approximates the set of abstract memory locations a pointer expression may reference. This information underlies a broad range of applications, including compiler optimizations [19], program slicing [21, 34], bug detection [11, 31, 32, 37–39, 42, 44], change-impact analysis [30], and program verification [1, 9, 10]. Because these clients often run in latency-sensitive pipelines such as code review and regression testing, the cost of pointer analysis translates directly into developer wait time. At the same time, higher precision strengthens downstream reasoning, such as reducing false alarms and unlocking more opportunities for compiler optimization.

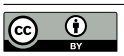
Building an effective program analyzer requires addressing both high-level design decisions (e.g., selecting a suitable abstraction for a downstream application) and low-level engineering concerns (e.g., designing and implementing efficient data structures). A persistent bottleneck in inclusion-based pointer analyses is the cost of manipulating *points-to sets*, the sets of abstract objects associated with pointer variables. Constraint propagation repeatedly performs operations such as union, difference, and subset testing over these sets. As the number of variables and abstract objects grows, set operations dominate the inner loop, motivating decades of work on compact representations and fast bulk operations [12, 40].

Existing representations span a wide spectrum. BDD-based designs offer compactness and efficient logical operations on certain workloads, but incur substantial constant factors and irregular memory-access patterns [40]. Bit-vector-based schemes provide predictable word operations and good cache locality; compression can further exploit the observation that most variables point to only a small fraction of all objects [12].

Sparsity, however, changes the nature of the bottleneck: once a set is encoded as a sequence of sparse blocks, many operations become dominated by *merge-like matching* of block indices (e.g., "do these two blocks contain the same range of the universe") rather than by bitmap arithmetic itself. This matching step is branch-heavy and typically involves pointer chasing, a pattern that modern CPUs handle poorly relative to regular vector computation. Recent work by Barbar and Sui [2, 3] has shown that points-to-set representations still admit substantial improvements, for example, through hash consing and object clustering. These optimizations reduce redundancy in the sets being propagated, but they do not address the low-level cost of repeatedly matching sparse block identifiers during set operations.

This paper argues that points-to-set representation permits further substantial gains because contemporary processors provide wide SIMD units capable of comparing and manipulating multiple values per instruction. Conventional sparse bit-vector kernels, however, rarely exploit SIMD: their critical path is the scalar, branch-driven process of advancing through mismatched block indices to locate matching blocks. Our goal is to reorganize sparse-set operations so that index matching becomes a regular, branch-free computation amenable to vectorization.

We present *IndexedBlockBitVector* (IBBV), a SIMD-aware sparse bit-vector design. First, we redesign the sparse bit-vector layout to



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837421

store block IDs and their associated bitmaps contiguously, exposing the sorted structure directly to SIMD operations (§ 3). Second, we build set operators around a SIMD-friendly index-matching primitive: rather than advancing one index at a time via conditional branches, each operator loads batches of block IDs from both operands, compares them in parallel, computes mask-based advance decisions, and applies the corresponding block-level operation. A single shared kernel implements union, intersection, difference, and subset testing while preserving sparsity (§ 4).

While there have been several efforts on parallelizing pointer analysis at different precision levels with GPUs [14, 26, 36] and CPUs [22, 35, 45], they address a different layer of the scalability problem by exploiting coarse-grained parallelism across graph traversals or constraint-processing tasks. In contrast, SIMD-based acceleration of pointer analysis remains comparatively underexplored. Besides, our work is complementary: we target the inner loop of sparse points-to-set operations, and thus may benefit solvers even when their outer propagation strategy is sequential or already parallelized.

We implement IBBV in Phoenix [43], the alias analysis component of the Lotus framework [18], and evaluate it on 12 real-world programs under context-insensitive, 1-CFA, and 2-CFA analyses, using three propagation strategies: Wave Propagation, Deep Propagation, and Partial Update Solver [23]. We compare against five baselines: BDD [40], LLVM's sparse bit-vector (SBV) [24], WAH [41], CONCISE [6], and Roaring [20]. Against SBV, our approach reduces end-to-end analysis time across all 12 benchmarks, with gains that grow with context sensitivity, reaching up to 3.8× speedups and 86% reduction in peak memory. IBBV also outperforms BDD, WAH, CONCISE, and Roaring. On several hard 2-CFA instances where most baselines time out, our implementation completes the analysis. These benefits hold consistently across all three constraint propagation strategies.

In summary, we make the following contributions:

- A SIMD-accelerated sparse bit-vector design for points-to sets that replaces branch-heavy index matching with parallel comparisons and mask-guided advancement.
- SIMD-aware algorithms for set operations that retain the benefits of sparsity while exploiting data-parallel hardware.
- An end-to-end evaluation across real-world programs and multiple pointer analysis variants, demonstrating substantial time and memory savings.

2 Background and Overview

This section reviews the set representations for inclusion-based pointer analysis (§ 2.1), explains why existing sparse bit-vector implementations are difficult to accelerate with SIMD (§ 2.2), and outlines our solution (§ 2.3).

2.1 Set Representations in Pointer Analysis

Inclusion-based Pointer Analysis. Whole-program pointer analysis computes, for each pointer variable p, a *points-to set*: an over-approximation of the abstract objects that p may reference at run time. In the inclusion-based (Andersen-style) formulation, the analysis reduces to a system of subset constraints over these sets (e.g., assignments induce inclusions between points-to sets, and dereferences induce inclusions through the objects being referenced).

A standard solver iteratively propagates newly discovered elements along the constraints until it reaches a global fixed point. Because each propagation can touch many points-to sets—and real-world code triggers *millions* of such incremental updates—the asymptotic and constant-factor costs of set operations (insertion, union, membership, etc.) dominate overall runtime. Consequently, the representation of points-to sets is one of the key determinants of scalability for the entire analysis.

Binary Decision Diagrams (BDDs). BDD-based pointer analysis represents sets using directed acyclic graphs that encode Boolean functions. Each path from the root to a terminal node represents a particular assignment of variables, corresponding to the presence or absence of objects in the set. BDDs offer canonical representation (equivalent sets have identical representations after reduction) and can exploit regularities in the data. However, their performance is highly sensitive to the ordering of variables—an NP-hard optimization problem—and their memory usage can grow unpredictably, making them challenging to configure for large-scale analyses.

Sparse Bit-Vector Representations. A dense bit-vector assigns each abstract object a fixed bit position in a length-n bitmap, enabling fast word-parallel operations but becoming space-prohibitive when the object universe is large. Sparse bit-vectors (SBVs) preserve bit-vector semantics while materializing only non-empty regions. A common design partitions the universe into fixed-size blocks; each occupied block is stored as a *block identifier* paired with a small bitmap payload. Set operations then reduce to a merge over two sorted streams of block identifiers: matching identifiers trigger bitmap arithmetic, while unmatched blocks are skipped, copied, or discarded depending on the operator.

2.2 Limitations of Existing SBVs for SIMD

SIMD (Single Instruction, Multiple Data) instructions apply a single operation to multiple data lanes in parallel. For example, x86 CPUs provide 256-bit AVX2 and 512-bit AVX-512 execution, enabling operations over 8× or 16× 32-bit integers per instruction.

Requirements for Effective SIMD Set Processing. Accelerating SBV set operations with SIMD requires two properties:

- *Contiguous, predictable data.* SIMD is effective when the elements processed together reside in contiguous memory, enabling wide loads/stores and effective prefetching.
- *Mask-based control.* SIMD ISAs support masked execution, allowing an implementation to compute match masks and apply updates only on specified lanes, replacing unpredictable branches with data-parallel, mask-guided computation.

Limitations of Conventional SBV Implementations. A representative baseline is LLVM's SparseBitVector [24], which stores blocks in a sparse hierarchy with pointer-linked nodes. This design emphasizes performance of random insertion, but it violates the two requirements above at the solver's hot path: traversal is pointer-serial rather than contiguous, and block matching is implemented as a branch-heavy merge over sparse identifiers.

This observation motivates our design: to benefit from SIMD, the representation must expose contiguous index streams and exploit regular, mask-driven matching.

2.3 IndexedBlockBitVector at a Glance

We propose a sparse bit-vector layout optimized for contiguous memory access, together with SIMD-aware kernels built around a shared index-matching core. The key idea is to turn the branch-heavy part of sparse-set processing, namely *matching sorted block identifiers*, into a branch-free, data-parallel primitive.

Bottleneck. Set operations on sparse bit-vectors follow a common pattern: before combining blocks, the solver must first identify which block identifiers appear in both operands. Consider two sparse bit-vectors A and B with the following structure:

- A: indexes $\{0, 64, 128, 256, 512\}$, each with a 32-bit block.
- B: indexes $\{0, 32, 64, 256\}$, each with a 32-bit block.

The standard union algorithm processes one index from each operand per iteration, comparing them to determine whether to advance A, advance B, or match and combine. For this example, the traversal follows the sequence 0=0, 64>32, 64=64, 128<256, and 256=256. This pattern is inherently branch-heavy, and the same matching logic recurs across union, intersection, difference, and subset checking.

The consequence is that the solver spends substantial time on control and traversal rather than on the bitmap operations themselves. This is precisely the portion of execution that conventional SBVs expose poorly to SIMD.

Representation. To enable SIMD acceleration, we introduce the IndexedBlockBitVector (IBBV). Unlike the pointer-based structure, IBBV stores block identifiers and their associated bitmaps in contiguous arrays (§ 3). This layout makes the hot traversal hardware-prefetcher-friendly and allows the implementation to load batches of identifiers and payloads directly into vector registers.

Shared Matching Kernel. The IBBV layout allows us to redefine index matching as a parallel search over small batches. At a high level, the implementation loads vectors of block identifiers from both operands, computes a match mask, and then applies the appropriate block-level operation only to the matched positions.

For example, on AVX-512 platforms, we can instantiate this kernel using the `VP2INTERSECT` instruction to identify equal indices across 16-element vectors. The resulting match mask drives both the per-block bitmap computation and a safe advance policy (Figure 4, Proposition 4.1), allowing the traversal to skip over non-matching regions in larger steps than a scalar merge loop.

This factoring gives the design a simple structure: all set operators share the same SIMD-matching skeleton, and only the per-block computation varies. Union uses OR, intersection uses AND, difference uses AND-NOT, and subset testing uses EQUALS checks. In this way, a single high-throughput matching core supports the full range of set-theoretic operations required by the pointer analysis.

Example 2.1 (SBV, Roaring, and IBBV by Example.). Consider a sparse points-to set containing object IDs $\{0, 1, 641\}$. With 128-bit blocks, IDs 0 and 1 fall into the block with base 0, while ID 641 falls into the block with base 640. An SBV stores these two non-empty blocks sparsely, but a typical linked representation still traverses

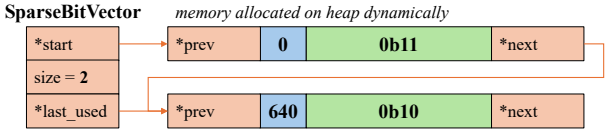


Figure 1: SparseBitVector (SBV) representation with bits at 0, 1, and 641 set.

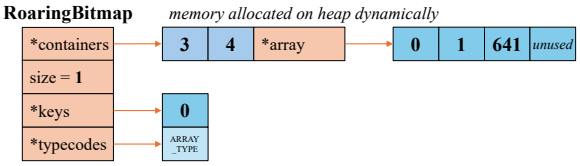


Figure 2: RoaringBitmap representation with bits at 0, 1, and 641 set. Some metadata and details (e.g. indirect pointers) are omitted for simplicity.

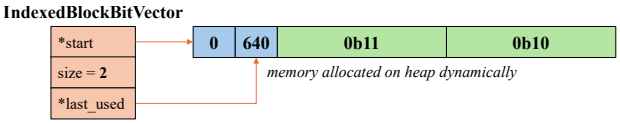


Figure 3: IndexedBlockBitVector (IBBV) representation with bits at 0, 1, and 641 set.

Table 1: SBV, Roaring, and IBBV containing object IDs $\{0, 1, 641\}$.

Aspect	SBV	Roaring	IBBV
Granularity	128 bits	2^{16}	128 bits
Layout	Linked nodes	Containers	Flat arrays
Example	2 blocks	1 container	2 blocks
Pointer chasing	✓	Partial	No

node pointers when matching two sets. Roaring instead places all three IDs in the same 2^{16} universe partition and stores them in an adaptive container, which is compact for this example but not organized around 128-bit block matching. IBBV stores the two block bases $[0, 640]$ and their payloads in flat arrays, preserving the same sparse block view as SBV while exposing contiguous indices to the SIMD matching kernel. Figure 1, 2 and 3 illustrate the representations with object IDs $\{0, 1, 641\}$. Table 1 summarizes the contrast.

3 SIMD-Friendly Set Representation

Our goal is not merely to compress points-to sets, but to expose them in a layout amenable to SIMD traversal. Concretely, the representation must satisfy three constraints simultaneously: (1) preserve sparsity, since points-to sets are typically small relative to the universe; (2) store block identifiers in contiguous, fixed-width runs so the solver can perform bulk match detection using vector loads and comparisons; and (3) support efficient in-place updates together with canonical storage, so fixed-point iteration can reuse stable, deduplicated sets without sacrificing update throughput.

To this end, we introduce the IndexedBlockBitVector (IBBV), a sparse-yet-contiguous representation tailored to the interface expected by our SIMD matching kernel. IBBV stores block identifiers

and associated payloads in packed arrays, exposing enough regularity to enable vectorized matching while preserving the solver-visible semantics of a canonical sparse bit-vector.

3.1 Block-Based Layout

IBBV partitions the universe into fixed-width blocks. Each block encodes a contiguous range of elements as a dense bitmap, and only blocks containing at least one set bit are stored. Concretely, IBBV maintains two parallel arrays: (1) an *index* array of block base positions (sorted ascending), and (2) a *block* array of 128-bit payloads. This layout defines a simple interface for the set operators: the index array exposes the sparse structure to the matching kernel, while the block array provides contiguous payloads for bulk bitmap operations. A block-base position is the smallest global element index covered by that block. This choice allows the block base to serve as a direct "address" into the bit-vector domain while keeping the representation sparse.

In our implementation, each block comprises 128 bits, and the kernels process four blocks per instruction using 512-bit AVX registers (i.e., 4×128 bits). Blocks are fixed-size even at the end of the logical object universe: if the final block covers fewer than 128 valid object IDs, the unused bits are kept zero and never reported as points-to facts. We maintain the index array in ascending order, which induces a merge-style traversal for bulk operations: the SIMD kernel compares multiple candidate block bases in parallel and, upon a match, applies bitwise operations to the corresponding payload vectors. This is the central advantage of IBBV: it retains sparsity while exposing precisely the regular structure required by the matching kernel.

Figure 3 depicts an IBBV in which bits $0, 1, 641$ are set. Bits 0 and 1 fall in the block with base index 0 (covering positions 0–127), whereas bit 641 falls in the block with base index 640 (covering positions 640–767). Accordingly, the sparse representation materializes exactly two entries: two block-base indices and their corresponding 128-bit payload blocks.

Example 3.1. Consider a points-to set with allocation sites at positions $\{0, 1, 5, 128, 130, 256, 257, 258, 1000, 1005, 1010\}$. With 128-bit blocks, the representation is:

Block Base	Block Range	Bits Set
0	0–127	0, 1, 5
128	128–255	0, 2 (global positions 128, 130)
256	256–383	0, 1, 2 (global positions 256, 257, 258)
896	896–1023	104, 109, 114 (global positions 1000, 1005, 1010)

Compared with pointer-linked sparse structures, IBBV removes per-node pointer overhead and improves spatial locality. Under a common configuration (128-bit blocks, 32-bit indices, two 64-bit pointers per node), a linked-list node requires 4 bytes for the index, 16 bytes for the bitmap payload, and 16 bytes for pointers. Excluding metadata, for the example above with four non-empty blocks, a linked-list representation consumes

$$4\times(4 \text{ bytes index}+16 \text{ bytes block}+16 \text{ bytes pointers}) = 144 \text{ bytes},$$

whereas IBBV uses

$$4 \times (4 \text{ bytes index} + 16 \text{ bytes block}) = 80 \text{ bytes}.$$

In addition to reducing memory footprint, the packed arrays permit higher effective memory bandwidth during scanning and bulk set operations. More importantly, they align the representation with the set operators' dominant access pattern: repeated scans over sorted sparse block IDs followed by block-wise updates on the matched positions.

3.2 Temporal Locality and Memory Alignment

Single-bit updates and membership tests are frequent in pointer analysis, and they often show near-sequential access: once a block becomes active, additional bits in the same block (or a neighboring block) tend to be queried or set soon afterward. IBBV leverages this behavior by caching the most recently accessed block position in a `last_used` pointer.

Given an operation at global bit position x, IBBV computes the block base $\lfloor x/128 \rfloor \times 128$ and first checks whether it matches the cached entry. If so, the update proceeds immediately. If not, it falls back to a binary search over the index array, using the cached position as the first boundary for comparison. This "check-last-used-first" strategy reduces comparisons during runs of updates that stay within a single block or move gradually across blocks.

Example 3.2. During propagation along an assignment chain $p_1 \leftarrow p_2 \leftarrow \cdots \leftarrow p_n$, new points-to facts are often discovered in clusters. Suppose that the bits i, i+1, i+2, ... are inserted in order. After the first insertion finds the block containing i, `last_used` caches that block's array position. Subsequent insertions frequently hit the cached block until the sequence crosses the next 128-bit boundary, at which point the `last_used` will be updated for subsequent accesses.

With the `last_used` field, the amortized time complexity for getting, setting, and resetting bits sequentially is $O(1)$ rather than the $O(\log n)$ cost of a binary search on every access.

We do not enforce explicit alignment for the `index` and `block` arrays. The merge-style scan advances by data-dependent offsets, so strict alignment would either introduce padding—inflating the representation—or add control flow to realign accesses. Contemporary CPUs handle unaligned SIMD loads with a small, typically acceptable penalty. We therefore use unaligned loads for data-dependent accesses and align only those temporary buffers whose layout we fully control, mitigating any residual overhead.

4 SIMD-Enhanced Set Operations

Sparse-set operations in pointer analysis reduce to a common core: given two sorted arrays of sparse block identifiers, identify the matching blocks and determine how far each side can advance without skipping a future match. In conventional sparse bit-vectors, this core is realized as a scalar merge loop with data-dependent branches. Algorithm 1 illustrates this scalar baseline.

We factor this core into a shared SIMD matching kernel. The kernel compares batches of block identifiers, derives safe advancement decisions from the resulting masks, and applies the block-level operation only at matched positions. The presentation below is ISA-neutral: we assume a SIMD primitive for parallel index intersection,

Algorithm 1: Sparse Bit-Vector Union (Scalar Version)

Input: Bit-vector A and B with block-based representation
Output: A boolean value indicating whether A is changed
Result: A is joined with B

```
1  indexA ← 0, indexB ← 0, indexC ← 0, changed ← false;
2  while indexA < A.size() and indexB < B.size() do
3      if A.indexes[indexA] < B.indexes[indexB] then
4          indexA ← indexA + 1;
5      else if A.indexes[indexA] > B.indexes[indexB] then
6          insert index and block at B[indexB] before A[indexA];
7          indexA ← indexA + 1, indexB ← indexB + 1;
8          changed ← true;
9      else
10         // compute bit-wise OR of two matched blocks
11         result ← A.blocks[indexA] | B.blocks[indexB];
12         changed ← changed | (result! = A.blocks[indexA]);
13         A.blocks[indexA] ← result;
14         indexA ← indexA + 1, indexB ← indexB + 1;
15 if indexB < B.size() then
16     // Concat the remaining elements of B to A
17     append index and block of B[indexB..B.size()] to A;
18     changed ← true;
19 return changed;
```

and express the remaining logic using standard SIMD comparisons, masked loads, and bitwise operations.

4.1 Parallel Index Intersection

The foundation of our vectorized approach is a batch-level answer to two questions: which loaded block identifiers already match, and how many identifiers can be consumed safely from each side. The second question is the core correctness challenge: simply intersecting the current SIMD windows is insufficient because an unmatched identifier in the current window of A may still match an identifier just beyond the current window of B, and vice versa.

SIMD Index Intersection. Here, we assume an abstract primitive SIMD_INTERSECT_U32 that compares two vectors of n ascending 32-bit indices and returns two n-bit match masks ($matchA$, $matchB$). Bit i in $matchA$ is set if the i-th index in A appears in the loaded indices from B; $matchB$ is defined symmetrically. Some processors provide this operation directly in hardware; otherwise, it can be implemented via an emulated fallback (e.g., compare-and-permute strategies) with a higher constant cost. Figure 4 illustrates the effect of this primitive.

Given the match masks, we must still decide how many elements to advance. Simply advancing all n elements on both sides is incorrect: an element in the current window of A may match an element just beyond the current window of B.

Computing the Advance Counts. Algorithm 2 computes safe advance counts after intersecting n indices from each operand (we use $n = 16$ in our implementation, but the method generalizes).

Let $maxA$ be the last loaded index from A and $maxB$ be the last loaded index from B. Any loaded $A[i] \le maxB$ cannot match a future B element beyond $maxB$, so such elements can be advanced safely; the same holds for B. We implement this idea by comparing

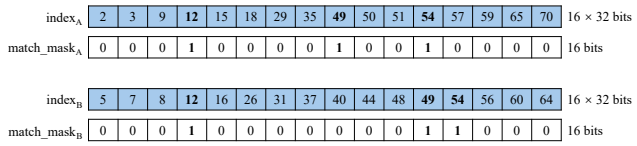


Figure 4: Index matching using SIMD parallel intersection. Two vectors of sorted integers are compared, producing bit masks that mark which integers from each side appear in the other. Bold elements are matched.

the loaded vectors against the opposing maximum and then turning the comparison masks into integer counts.

Proposition 4.1 shows that the number of elements that we can safely advance has a guaranteed lower bound. In our configuration, which intersects 16 indices from each operand, at least 16 elements can be advanced in total.

Proposition 4.1 (Minimal Advance Count). *For two strictly ascending arrays A and B, if we intersect n elements from each array per iteration, then the safe advance counts satisfy:*

$$k_A + k_B \ge n.$$

Proof. Merge-sort $A_1, \ldots, A_n, B_1, \ldots, B_n$ into $C_1, \ldots, C_{2n}$. Let the one-based position of $\min\{A_n, B_n\}$ in C be m. The order of identical elements does not matter.

Case 1: $m < n$. Impossible since A_n is greater than at least $n-1$ elements $(A_1, \ldots, A_{n-1})$, and so is B_n.

Case 2: $m \ge n$. There are m elements such that

$$C_i \le A_n \text{ and } C_i \le B_n \quad (i = 1, \ldots, m).$$

These elements can be advanced. Since $m \ge n$, at least n elements can be advanced in total.

□

4.2 Vectorized Union Operation

Union is central to whole-program points-to analysis and often dominates runtime. Using the shared matching kernel, each vectorized union iteration has five phases: (1) compute match masks, (2) compute safe advance counts, (3) scatter matched blocks from B to SIMD-ready positions, (4) apply masked SIMD OR operations, and (5) store OR result back to A.

Algorithm 2: SIMD-Based Advance Count

Input: Two arrays A and B, each containing at least n ascending uint32 elements
Output: k_A and k_B, indicating how many elements can be advanced respectively

```
1  vecIdxA ← LOAD(A), vecIdxB ← LOAD(B);
2  vecMaxA ← SIMD_SET1_U32(A[n − 1]);
3  vecMaxB ← SIMD_SET1_U32(B[n − 1]);
4  leMaskA ← SIMD_CMPLE_U32(vecIdxA, vecMaxB);
5  leMaskB ← SIMD_CMPLE_U32(vecIdxB, vecMaxA);
6  k_A ← n− LZCNT_U32(leMaskA);
7  k_B ← n− LZCNT_U32(leMaskB);
8  return k_A, k_B;
```

Block Scattering. The key challenge is data alignment for SIMD OR. Because indices in A and B are sparse and not position-aligned, we cannot perform an OR operation directly on consecutive block chunks. Instead, we scatter matched blocks from B into a temporary buffer at positions aligned with A (Figure 5), and append unmatched blocks to buffer C for later merge.

Algorithm 3 illustrates the full union flow. Each iteration computes match masks and advance counts, scatters the advanced portion of B into either `tmp` (matched) or C (unmatched), and then applies SIMD OR to 4-block chunks, 4 times. Masked loads treat unmatched `tmp` entries as zero, so only true matches contribute to the OR result.

The vectorized loop only runs while both operands have at least one full SIMD batch of block identifiers remaining. When either side has fewer than n remaining identifiers, the implementation falls back to the scalar routine shown in Algorithm 1.

Example 4.2. Consider computing $A \cup B$, where A contains blocks at indexes {10, 20, 30, 40, **42**, **49**, 50, 60, **63**, 70, …, 130} and B contains blocks at {15, 25, 35, **42**, 45, **49**, 55, **63**, 65, …, 135}. For simplicity, a block indexed at 10 is the eleventh block in the universe, with block identifier 1280, containing bits 1280–1407.

In one vectorized iteration, SIMD_INTERSECT_U32 identifies matches at indexes **42**, **49**, and **63**. The advance counts allow A to advance by 16 and B to advance by 15. The scattering phase copies the three matched blocks from B into `tmp` at offsets aligned with their partners in A, and appends the other advanced blocks from B into C. The SIMD OR phase then processes 4-block chunks, masking out all `tmp` lanes except the matched ones.

4.3 Other Set Operations

Once matching and safe advance count are factored into a shared kernel, the remaining set operators differ only in their block-level semantics and in how they materialize the result.

Set Intersection and Difference. For intersection and difference, the number of blocks never increases. We write non-zero results in place and compact away zeroed blocks during the vectorized traversal. Concretely, matched blocks use AND (set intersection) or AND-NOT (set difference), followed by SIMD compressed store to keep only non-empty blocks, and no separate cleanup pass is required. We remove empty blocks to ensure a canonical representation and to speed up equality testing, single-bit access, and all vectorized operations.

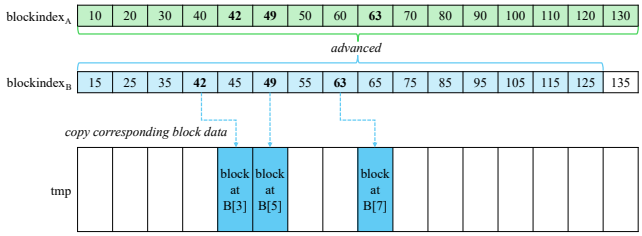


Figure 5: Block scattering for vectorized union. Matched blocks from B are scattered into a temporary buffer (indexed by matched positions in A) so the subsequent SIMD OR can run on dense 4-block chunks; unmatched blocks from B are appended to buffer C for later merge into A.

Algorithm 3: SIMD-Based In-Place Union Operation

Input: Bit-vectors A and B with block-based representation
Output: A boolean value indicating whether A is changed
Result: A is joined with B

$indexA \leftarrow 0, indexB \leftarrow 0, indexC \leftarrow 0$;
$changed \leftarrow$ **false**;
while $indexA + n \le A.size()$ **and** $indexB + n \le B.size()$ **do**
  $vecIdxA \leftarrow$ LOAD(&$A.indexes[indexA]$), $vecIdxB \leftarrow$ LOAD(&$B.indexes[indexB]$);
  $matchA, matchB \leftarrow$ SIMD_INTERSECT_U32($vecIdxA$, $vecIdxB$);
  $advA, advB \leftarrow$ ADVANCE_COUNT(&$A.indexes[indexA]$, &$B.indexes[indexB]$);
  /* For each advanced block of B: copy to the corresponding position for SIMD computation if index matched, append to C otherwise. Blocks not advanced are ignored. */
  SCATTER_OR_APPEND(*tmp,A,B,C,indexA,indexB,indexC, matchA,matchB,advA,advB*)
  /* Compute bit-wise OR using vector registers. Unmatched blocks in tmp are masked to zero. */
  for $i \leftarrow 0$ **to** $n - 4$ **step by** 4 **do**
    $vecBlkA \leftarrow$ LOAD(&$A.blocks[indexA + i]$);
    $vecBlkB \leftarrow$ LOAD_MASKZ_U128(&$tmp[i]$, $(matchA \gg i)\&0xF$);
    $result \leftarrow$ SIMD_OR($vecBlkA$, $vecBlkB$);
    $changed \leftarrow changed \mid (result \neq vecBlkA)$;
    STORE(&$A.blocks[indexA + i]$, $result$);
  $indexA \leftarrow indexA + advA, indexB \leftarrow indexB + advB$;
/* Scalar computation for the remaining elements. It updates A in place and appends additional elements to C. */
SCALAR_UNION($A, B, C, indexA, indexB, indexC, changed$);
if $indexC > 0$ **then**
  merge-sort A, C into A;
  $changed \leftarrow$ **true**;
if $indexB < B.size()$ **then**
  /* Unprocessed elements in B are the largest, no comparison needed */
  append index and block of B[indexB..B.size()] to A;
  $changed \leftarrow$ **true**;
return $changed$;

Table 2: Characteristics of vectorized set operations. All operations share the same index matching logic but differ in block operations and post-processing.

Operation	Bitwise Op	Post-processing
Union	OR	Merge buffer C when finalizing
Intersection	AND	Compact zeroed blocks
Difference	AND-NOT	Compact zeroed blocks
Subset test	AND, EQUALS	Early return on failure

Subset Testing. Given sets A and B, we decide whether $A \subseteq B$ by (i) ensuring that every block present in A is also present in B, and (ii) verifying, for each matched block, that the corresponding bitmap satisfies $A.block \wedge B.block = A.block$. The vectorized implementation checks both conditions in each iteration and short-circuits, returning `false` as soon as either condition is violated.

Membership Testing and Updating. Block identifiers are maintained in a sorted index array; thus, membership reduces to binary search. The operation `test(`i`)` locates, in $O(\log n)$ time, the block that would contain i (where n is the number of blocks) and then reads the corresponding bit.

Updates that preserve the block set follow the same locate-then-update scheme. Hence, `set(`i`)` and `reset(`i`)` run in $O(\log n)$ worst-case time when they only modify bits within an existing block.

When an update changes the block set—either by clearing the last remaining bit of a block or by setting a bit in a previously absent block—the representation must delete or allocate a block and splice the index array accordingly. These structural changes may require shifting $\Theta(n)$ array elements, resulting in $O(n)$ worst-case time.

Equality Test and Cloning. Our representation is canonical: indices are sorted, and all blocks are non-empty. As a result, each set has a unique memory layout, and two sets are equal iff their index and block arrays are byte-wise identical. Equality testing, therefore, checks the array lengths and then performs a bulk comparison over the contiguous buffers (e.g., `memcmp`).

The cloning operation is analogous. Because the representation is stored in flat arrays, we can duplicate a set via bulk copy (e.g., `memcpy`) rather than allocating and copying per-node structure. These operations are critical in dataflow-style analyses: equality (and subset) checks drive fixed-point detection, while cloning is used to initialize per-iteration state.

5 Evaluation

We evaluate our optimized set representation by addressing the following research questions:

- **RQ1**: How does IBBV compare with set implementations already used in pointer analysis?
- **RQ2**: How does IBBV compare with other compressed and SIMD-enhanced bitset implementations?
- **RQ3**: Does IBBV remain effective under different constraint-solving strategies in pointer analysis?

Evaluation Scope. Our study is based on the inclusion-based pointer analyses for C/C++ implemented in the Lotus program analysis framework [18, 43]. The machine has AVX-512 support, so the measured speedups should be read as evidence for this setting rather than as a claim about all pointer-analysis frameworks or all processor families. The representation itself is language-independent because it stores abstract object IDs, but the measured benefit depends on whether the analysis repeatedly manipulates sparse points-to sets with stable integer object IDs and whether those operations constitute a significant fraction of the end-to-end runtime.

Benchmarks. Our benchmark suite consists of 12 real-world C/C++ programs from diverse application domains, as shown in Table 3. These programs span multiple domains and a wide range of sizes and complexities, from small utilities like *FileCheck* (45K LoC) to large-scale systems such as *php* (2.8M LoC). This variation stresses both scalability and memory efficiency, ensuring that our evaluation captures performance characteristics across varying program scales and configurations.

Table 3: Benchmark suite characteristics. Lines of code (LoC) are obtained using cloc.

Program	Description	LoC	Functions	Call Sites
x264	video encoding library	72.9K	0.9K	4.8K
tmux	terminal multiplexer	74.3K	2.6K	13.5K
povray_r	ray-tracing software	81.2K	2.1K	17.7K
zsh	unix shell	133.1K	1.4K	11.0K
bash	shell program	418.4K	3.4K	19.6K
sqlite3	relational database	242.2K	2.6K	14.1K
php	language interpreter	2848.6K	10.5K	122.1K
phpdbg	PHP debugger	2848.6K	10.8K	141.2K
bssl	cryptographic library	520.0K	12.4K	45.0K
FileCheck	LLVM test utility	45.0K	14.2K	45.7K
llvm-mt	LLVM manifest tool	45.0K	12.1K	38.8K
yara	pattern matching utility	85.0K	1.2K	8.4K

Baselines. We compare our SIMD-optimized approach (IBBV) against several alternative set representations:

- *Sparse Bit-Vector (SBV)*: A linked-list-based sparse bit-vector implementation from the LLVM project, similar to those found in many pointer analysis frameworks [24].
- *Binary Decision Diagrams (BDD)*: A BDD-based set representation using the CUDD library [33].
- *Word-Aligned Hybrid (WAH) bitmap*: Word-aligned run-length bitmap compression using literal and fill words for efficient logical operations [41].
- *Compressed 'n' Composable Integer Set (CONCISE)*: Improved WAH compression, reduces wasted space while keeping composable bitwise operations [6].
- *Roaring Bitmap*: A hybrid implementation that splits integers into blocks identified by their high 16 bits and uses adaptive containers [20].

Analyses. We evaluate IBBV on the following inclusion-based pointer analysis variants: (1) *Andersen's analysis*: a flow-insensitive, context-insensitive inclusion-based analysis; (2) *context-sensitive Andersen-style analysis*: a flowinsensitive, contextsensitive inclusion-based analysis. We instantiate two k-limiting configurations, 1-CFA and 2-CFA. To ensure a fair comparison, all variants are implemented within Phoenix [43], sharing identical algorithms and differing only in their underlying set representations.

Environment. We conduct the experiments on an AMD EPYC 9754 processor with 1024GB DDR5 RAM, running Ubuntu 22.04 LTS. The `VP2INTERSECT` is not implemented on our hardware, therefore an emulated fallback [7] is used. All implementations are compiled with GCC 11.4 using the -O3 optimization level and architecture-specific flags (-march=native). To ensure fairness, we disable CPU frequency scaling and maintain consistent thermal conditions across runs. Each benchmark is allowed 3 hours of runtime per run.

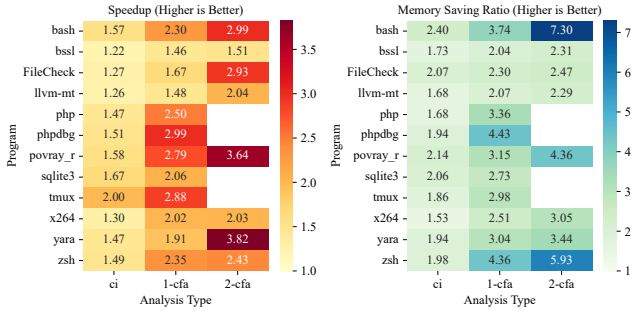


Figure 6: Speedup and memory saving ratio of IBBV compared to SBV using the wave-propagation solver.

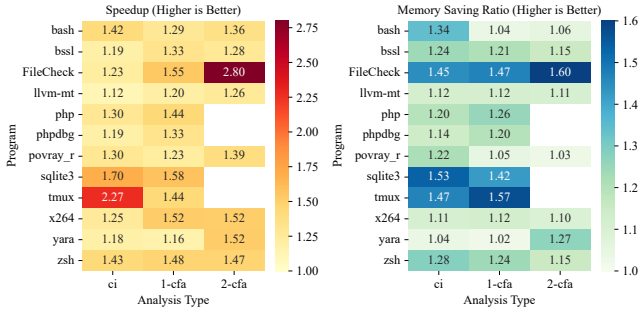


Figure 7: Speedup and memory saving ratio of IBBV compared to Roaring using the wave-propagation solver.

5.1 Comparison with Existing Representations in Pointer Analysis (RQ1)

We compare IBBV against the set representations most commonly used in pointer analysis, namely SBV and BDD. Table 4 reports the end-to-end runtime and peak memory usage. Table 5 summarizes the same raw measurements using geometric means. Across completed paired runs, IBBV outperforms SBV in both runtime and memory, and the runtime gap widens with increasing context sensitivity. The comparison with Roaring is intentionally separated because Roaring is the strongest general-purpose compressed bitmap baseline: IBBV's advantage is smaller than against SBV, but remains consistent across all analysis configurations.

Compared with BDDs. BDDs retain a modest memory advantage over IBBV on a few configurations (e.g., *bash* 1-CFA, *povray_r* 2-CFA, and *zsh* 2-CFA), consistent with their ability to exploit regularity in certain points-to relations. In runtime, however, they are typically an order of magnitude slower than IBBV. These results suggest that, in our whole-program setting, the constant costs of BDD manipulation are difficult to amortize despite their compact symbolic representation. This trend also differs from prior reports that BDD-based analyses trade moderate runtime overhead for substantial memory savings [13]. In our evaluation, the runtime gap is larger, while the memory advantage is much smaller. One plausible explanation is that bitmap-based representations benefit more directly from contemporary hardware features such as cache-friendly scans and SIMD bitwise operations, whereas BDD operations involve more complex pointer-rich manipulations. In addition, the larger and more heterogeneous benchmarks in our study may offer fewer regularities for BDD compression to exploit.

Compared with SBV. IBBV consistently outperforms SBV on runtime, and the gap widens as the analysis becomes more precise. For context-insensitive analysis, IBBV is faster on all 12 programs, typically by 1.2–1.5×. The improvement becomes larger for 1-CFA, where IBBV reduces runtime from 1304s to 454s on *tmux* (2.9×), from 883s to 295s on *phpdbg* (3.0×), and from 169s to 61s on *povray_r* (2.8×). Under 2-CFA, the gains are larger still on the benchmarks completed by both systems: for example, IBBV reduces runtime from 6895s to 2303s on *bash* (3.0×), from 5495s to 1511s on *povray_r* (3.6×), and from 1840s to 757s on *zsh* (2.4×). The difference is even more pronounced on the hardest workloads. SBV times out on the 2-CFA configurations of *php*, *phpdbg*, *tmux*, and *sqlite3*, whereas IBBV completes the former three within the time budget, in 4337s, 6788s, and 4788s, respectively.

IBBV also improves memory behavior relative to SBV, with the effect again growing on harder configurations. In the 2-CFA runs completed by both systems, IBBV reduces peak memory from 376.0GB to 51.5GB on *bash* (86%), from 177.6GB to 40.8GB on *povray_r* (77%), and from 127.8GB to 21.6GB on *zsh* (83%). More broadly, the 1-CFA and 2-CFA results show that IBBV typically reduces SBV's memory footprint by roughly half to three-quarters, consistent with IBBV's lower pointer-chasing overhead.

> **Answer to RQ1:** IBBV consistently outperforms the most relevant pointer analysis baselines, reducing both runtime and peak memory usage, with the largest benefits for precise and large-scale analyses, where it also converts several timeout cases into completed runs.

5.2 Comparison with Other Compressed Bitset Implementations (RQ2)

In this section, we compare IBBV with two classes of compressed bitsets: WAH [41] and CONCISE [6], which are run-length-oriented compressed bitmaps, and Roaring [20], a hybrid compressed bitmap with SIMD-aware optimizations.

Compared with WAH and CONCISE. WAH and CONCISE are consistently slower than IBBV, and their scalability deteriorates more sharply on the larger analyses. Even on completed runs, the gap is substantial. For example, on 1-CFA *bash*, IBBV requires 210s whereas WAH and CONCISE require 909s and 1016s; on 1-CFA *php*, IBBV requires 334s whereas WAH and CONCISE require 1103s and 1259s; and on 2-CFA *yara*, IBBV requires 86s while WAH and CONCISE require 260s and 257s. On several 2-CFA workloads, including *bash*, *povray_r*, and *tmux*, both representations time out, whereas IBBV still completes successfully.

Compared with Roaring. Roaring is the strongest general-purpose competitor, but IBBV still achieves clear gains across all analysis variants. In context-insensitive runs, IBBV is already faster across all benchmarks, reducing runtime from 142s to 62s on *tmux*, from 86s to 51s on *sqlite3*, and from 46s to 36s on *php*. The gap persists in the more precise analyses. On 1-CFA *sqlite3*, IBBV requires 435s whereas Roaring requires 689s; and on 1-CFA *phpdbg*, 295s versus 392s. Under 2-CFA, IBBV completes *bash* in 2303s whereas Roaring

Table 4: Performance evaluation using the wave-propagation solver. Execution time is reported in seconds; peak memory consumption is reported in GB. We display speedup and memory-saving ratio for IBBV and Roaring, both compared with SBV.

Program	Analysis	SBV		IBBV		BDD		Roaring		WAH		CONCISE	
		time	mem	time	mem	time	mem	time	mem	time	mem	time	mem
bash	CI	38	2.4	**24 (1.6x)**	**1.0 (2.4x)**	588	1.3	34 (1.1x)	1.4 (1.7x)	84	1.1	88	**1.0**
	1-CFA	482	25.3	**210 (2.3x)**	6.8 (3.7x)	3352	6.1	270 (1.8x)	7.0 (3.6x)	909	6.6	1016	**5.6**
	2-CFA	6895	376.0	**2303 (3.0x)**	**51.5 (7.3x)**	OOT	/	3141 (2.2x)	54.6 (6.9x)	OOT	/	OOT	/
bssl	CI	11	0.9	**9 (1.2x)**	**0.5 (1.8x)**	92	0.7	11 (1.0x)	0.7 (1.3x)	15	**0.5**	15	**0.5**
	1-CFA	31	2.2	**21 (1.5x)**	**1.1 (2.0x)**	196	1.3	28 (1.1x)	1.3 (1.7x)	50	**1.1**	55	**1.1**
	2-CFA	77	5.2	**51 (1.5x)**	2.3 (2.3x)	443	2.5	65 (1.2x)	2.6 (2.0x)	127	2.2	129	**2.1**
FileCheck	CI	50	2.1	**39 (1.3x)**	**1.0 (2.1x)**	782	1.6	48 (1.0x)	1.5 (1.4x)	79	1.1	82	**1.0**
	1-CFA	175	5.8	**105 (1.7x)**	2.5 (2.3x)	3389	3.5	162 (1.1x)	3.7 (1.6x)	283	2.5	314	**2.4**
	2-CFA	1816	24.4	**619 (2.9x)**	9.9 (2.5x)	OOT	/	1735 (1.0x)	15.8 (1.5x)	3084	10.3	3266	**9.8**
llvm-mt	CI	6	0.6	**5 (1.2x)**	**0.3 (2.0x)**	38	0.4	6 (1.0x)	0.4 (1.5x)	7	**0.3**	7	**0.3**
	1-CFA	22	1.1	**15 (1.5x)**	**0.5 (2.2x)**	141	0.6	18 (1.2x)	0.6 (1.8x)	23	**0.5**	24	**0.5**
	2-CFA	112	2.9	**55 (2.0x)**	1.3 (2.2x)	752	1.3	69 (1.6x)	1.4 (2.1x)	134	1.3	138	**1.2**
php	CI	52	3.0	**36 (1.4x)**	**1.8 (1.7x)**	357	2.1	46 (1.1x)	2.1 (1.4x)	87	**1.8**	93	**1.8**
	1-CFA	835	24.8	**334 (2.5x)**	7.4 (3.4x)	3156	8.9	482 (1.7x)	9.3 (2.7x)	1103	7.4	1259	**7.1**
	2-CFA	OOT	/	**4337 (≥2.5x)**	**64.8**	OOT	/	6317 (≥1.7x)	74.9	OOT	/	OOT	/
phpdbg	CI	58	2.8	**38 (1.5x)**	**1.4 (2.0x)**	252	1.6	46 (1.3x)	1.6 (1.7x)	55	**1.4**	55	**1.4**
	1-CFA	883	26.7	**295 (3.0x)**	6.0 (4.5x)	2812	6.9	392 (2.3x)	7.2 (3.7x)	640	5.8	697	**5.6**
	2-CFA	OOT	/	**6788 (≥1.6x)**	**62.8**	OOT	/	9146 (≥1.2x)	73.7	OOT	/	OOT	/
povray_r	CI	10	1.0	**6 (1.7x)**	**0.5 (2.0x)**	69	0.6	8 (1.3x)	0.6 (1.7x)	10	**0.5**	11	**0.5**
	1-CFA	169	10.3	**61 (2.8x)**	3.3 (3.1x)	449	3.4	75 (2.3x)	3.4 (3.0x)	245	3.1	263	**2.9**
	2-CFA	5495	177.6	**1511 (3.6x)**	40.8 (4.4x)	8408	**37.9**	2106 (2.6x)	42.2 (4.2x)	OOT	/	OOT	/
sqlite3	CI	85	3.8	**51 (1.7x)**	**1.8 (2.1x)**	1092	2.6	86 (1.0x)	2.8 (1.4x)	230	1.9	240	**1.8**
	1-CFA	898	34.9	**435 (2.1x)**	12.8 (2.7x)	7007	17.2	689 (1.3x)	18.2 (1.9x)	2009	12.6	2156	**11.6**
	2-CFA	OOT	/	OOT	/	OOT	/	OOT	/	OOT	/	OOT	/
tmux	CI	125	5.2	**62 (2.0x)**	**2.8 (1.9x)**	1219	3.7	142 (0.9x)	4.1 (1.3x)	291	**2.8**	306	**2.8**
	1-CFA	1304	33.1	**454 (2.9x)**	11.1 (3.0x)	5085	14.7	653 (2.0x)	17.5 (1.9x)	2348	10.6	2512	**10.1**
	2-CFA	OOT	/	**4788 (≥2.3x)**	**101.3**	OOT	/	7676 (≥1.4x)	129.7	OOT	/	OOT	/
x264	CI	3	0.4	**2 (1.5x)**	**0.3 (1.3x)**	6	0.4	3 (1.0x)	**0.3 (1.3x)**	**2**	**0.3**	**2**	**0.3**
	1-CFA	44	4.3	**22 (2.0x)**	1.7 (2.5x)	56	2.0	33 (1.3x)	1.9 (2.3x)	35	1.7	36	**1.6**
	2-CFA	268	23.3	**132 (2.0x)**	7.6 (3.1x)	304	7.9	201 (1.3x)	8.4 (2.8x)	294	7.6	300	**7.4**
yara	CI	2	0.4	**1 (2.0x)**	**0.2 (2.0x)**	3	**0.2**	2 (1.0x)	**0.2 (2.0x)**	2	**0.2**	2	**0.2**
	1-CFA	17	2.2	**9 (1.9x)**	**0.7 (3.1x)**	21	0.8	10 (1.7x)	**0.7 (3.1x)**	12	**0.7**	12	**0.7**
	2-CFA	328	13.2	**86 (3.8x)**	3.8 (3.5x)	415	4.3	131 (2.5x)	4.9 (2.7x)	260	3.6	257	**3.5**
zsh	CI	8	0.8	**5 (1.6x)**	**0.4 (2.0x)**	79	0.5	7 (1.1x)	0.5 (1.6x)	13	**0.4**	14	**0.4**
	1-CFA	120	10.5	**51 (2.4x)**	2.4 (4.4x)	522	3.1	75 (1.6x)	3.0 (3.5x)	142	2.4	155	**2.2**
	2-CFA	1840	127.8	**757 (2.4x)**	21.6 (5.9x)	4719	20.3	1115 (1.7x)	24.9 (5.1x)	3264	19.8	3540	**17.8**

Table 5: Consolidated geometric-mean speedup and memory-saving ratio of IBBV over SBV and Roaring. Each entry uses only paired configurations where both compared representations are complete.

Analysis	IBBV vs. SBV		IBBV vs. Roaring		Paired Runs	
	Time	Mem	Time	Mem	SBV	Roaring
CI	1.54×	1.92×	1.45×	1.27×	12	12
1-CFA	2.14×	2.98×	1.36×	1.22×	12	12
2-CFA	2.56×	3.54×	1.51×	1.18×	8	11

requires 3141s, *povray_r* in 1511s versus 2106s, and *zsh* in 757s versus 1115s. Roaring often has competitive memory usage, but it remains consistently slower than IBBV on these workloads.

Answer to RQ2: Compared with general-purpose compressed and SIMD bitmap libraries, IBBV delivers better end-to-end performance because it is tailored to the sparse block-index matching and update patterns that dominate pointer analysis solvers.

5.3 The Impact of Different Set Constraint Solving Strategies (RQ3)

In the preceding experiments, we use the Wave Propagation solver exclusively. To address RQ3, we evaluate IBBV under additional constraint-solving strategies. Because different strategies induce different mixes of set operations and exhibit different convergence behavior, this evaluation tests whether the benefits of IBBV are robust across solver schedules:

- *Deep Propagation (DP)*: Processes constraints until convergence, potentially revisiting constraints more aggressively. This strategy increases the frequency of subset checks and smaller incremental updates.
- *Partial Update Solver (PUS)* [23]: Reprocesses only the portion of the constraint graph affected by newly discovered facts. This strategy typically reduces redundant propagation, but introduces a more heterogeneous mix of unions, subset checks, and small incremental updates.

We mainly focus on 2-CFA, which leads to the most challenging workloads. The main result is that IBBV remains effective across all solver schedules, although the size of the gain varies with the operation mix. Wave propagation delivers strong improvements, which is consistent with its union-heavy execution pattern and with

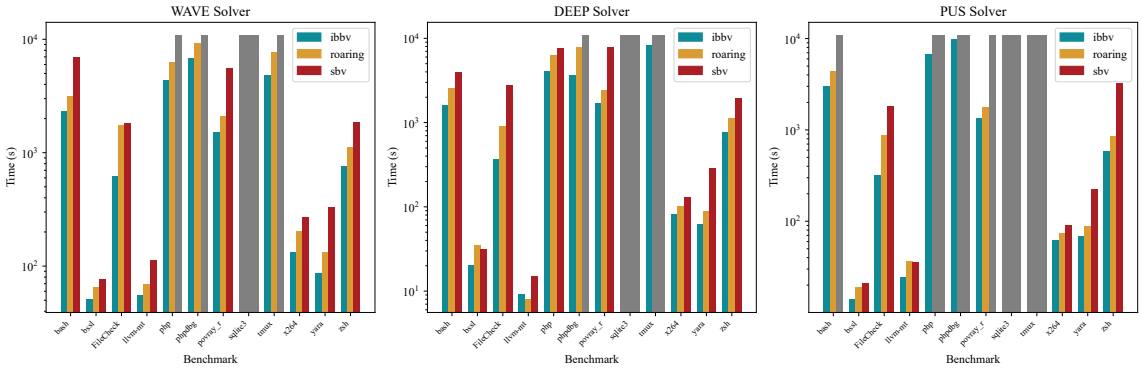


Figure 8: Runtime comparison across constraint solving strategies. IBBV maintains competitive performance across wave propagation, deep propagation, and the partial update solver. Results are reported for 2-CFA; timed-out cases are shaded gray.

the fact that union is the operation on which IBBV can amortize index matching most aggressively.

Deep propagation also yields substantial improvements. Because DP revisits constraints more aggressively, performs more subset checks, and makes smaller incremental updates, its operational mix differs from that of wave propagation; nevertheless, the results still show that the change in representation remains beneficial beyond a single union-heavy schedule.

The results for PUS show the same qualitative trend. Because PUS avoids some redundant propagation, its absolute runtime is lower than that of wave propagation on several benchmarks. But the choice of representation still matters, as the solver continues to spend substantial time on sparse set manipulation.

Overall, we find that WP, DP, and PUS results reveal the same trend: changing the propagation schedule affects how much work the solver performs, but it does not eliminate the importance of efficient sparse-set kernels.

> **Answer to RQ3:** IBBV remains effective across different propagation schedules, achieving the largest gains under union-heavy wave propagation but still delivering substantial speedups under deep propagation and PUS, demonstrating that its benefit is robust to solver strategy.

6 Discussions

6.1 Interpreting the Results

What the Baseline Comparisons Reveal. Relative to SBV, IBBV shows that a sparse bit-vector representation can remain solver-friendly while eliminating much of the pointer chasing and branch-heavy control flow on the hot path. Relative to Roaring, WAH, and CONCISE, our results suggest that general-purpose compression quality alone is not decisive: a representation that is efficient in isolation can still be a poor fit for the access patterns induced by points-to propagation. Relative to BDDs, the results support a complementary conclusion: while canonical symbolic structure can help on some workloads, in our setting, the constant factors and scalability limits of BDD manipulation are difficult to amortize. Taken together, the decisive factor is not whether a set representation is sophisticated in the abstract, but whether it aligns with the solver's dominant execution pattern.

Why the Gains Increase on Harder Analyses. The results exhibit a clear scaling trend: the benefit of IBBV grows from context-insensitive analysis to 1-CFA, and again to 2-CFA. This pattern indicates that IBBV accelerates precisely the component of the computation whose cost scales with precision—namely, repeated propagation over larger, more numerous sparse points-to sets. Put differently, higher precision amplifies the exact kernel that IBBV optimizes. This explains both why the largest relative gains appear on the most demanding configurations and why several previously timed-out cases become feasible under IBBV. A further distinction is worth noting: IBBV improves both runtime and memory footprint simultaneously, whereas the other representations we evaluate generally improve one dimension at the expense of the other. Looking forward, we believe IBBV can be combined with complementary optimizations, such as hash consing for caching [3].

6.2 Scope of the Evaluation

Targeted Pointer Analyses. Our evaluation focuses on exhaustive pointer analysis for C/C++, in particular Andersen-style analyses used either as stand-alone analyses or as a preprocessing phase for downstream clients. We do not currently target tightly interleaved, multi-domain analyses that propagate points-to information alongside other abstract domains (e.g., integer ranges, typestates) within a single, fused solver loop. In such pipelines, IBBV can still accelerate the aliasing component, but it may leave other dominant costs unchanged, thereby yielding limited end-to-end speedups. Moreover, integrating IBBV into these solvers would require substantially more engineering effort, both to expose suitable update interfaces and to coordinate with their scheduling and invariants.

Portability Across SIMD ISAs. The evaluated implementation uses AVX-512 because our largest workloads require hundreds of gigabytes of memory, and the available high-memory server in our environment supports AVX-512. The algorithmic structure, however, is not tied to 512-bit registers: IBBV uses sorted contiguous index arrays, vector comparisons, masks, and bitwise operations. On AVX2, the same matching kernel can be instantiated with narrower 256-bit vectors, processing fewer indices per batch. On ARM

NEON or SVE, the same assumptions apply if the implementation provides masked vector operations and shuffle across lanes; the main uncertainty is the constant factor of emulating the parallel index-intersection primitive when the ISA lacks a direct equivalent, as well as the cost of SIMD instructions when fewer elements are processed per iteration.

6.3 Threats to Validity

Internal Validity. To isolate the effect of the set representation, all competing representations are embedded in the same Phoenix solver, share the same constraint graph and worklist scheduling, and run on the same hardware; only the set data structure and its kernel differ. However, because the IBBV layout and the SIMD kernel are co-designed, we cannot cleanly attribute the gains to each component in isolation. The flat index and block arrays reduce pointer overhead, improve spatial locality, and make scans more amenable to hardware prefetching even without SIMD-specific matching. The SIMD kernel then provides an additional runtime benefit by replacing the branch-heavy scalar merge loop with parallel index comparisons and mask-guided advancement. This suggests that the memory reductions mainly follow from the layout change, while the runtime improvements reflect both improved locality and SIMD matching. A clean decomposition, however, is not equivalent to simply disabling vector instructions, since that would also change code generation, memory traffic, and scalar fallback behavior. To limit measurement noise, each configuration is executed with CPU frequency scaling disabled and under consistent thermal conditions.

External Validity. Our benchmark suite comprises 12 real-world C/C++ programs evaluated under three variants of inclusion-based analysis. This sample necessarily covers only a subset of the possible workload regimes; on programs where the sets involved are consistently small, or where the end-to-end runtime is dominated by phases outside points-to propagation, the relative benefit of IBBV should be expected to diminish. Our implementation is also embedded within a single solver (Phoenix) on a single hardware platform. Other frameworks may make different engineering trade-offs in worklist scheduling, constraint representation, or on-the-fly graph updates, and these choices could interact with the set representation in ways our experiments do not exercise.

7 Related Work

Set Optimizations for Pointer Analysis. Set operations are fundamental to many program analyses, and different representations expose different performance trade-offs. BDDs have long been used to represent points-to relations compactly in context-sensitive analyses [5, 40]. More recent work shows that points-to-set performance can also be improved by reducing redundancy inside the sets themselves. Barbar and Sui [3] canonicalize identical points-to sets and memoize repeated unions via hash consing, while subsequent work reduces points-to-set costs through object clustering [2] and object versioning [4]. These techniques are complementary to our approach: they reduce redundant set content or repeated work at the analysis level, whereas we accelerate the low-level sparse-set kernels that remain on the critical path during propagation. Beyond pointer analysis specifically, sparse dataflow frameworks also show that exploiting sparsity can substantially improve analysis efficiency [25]. Related work has also improved pointer-analysis scalability by adapting analysis precision to the program under analysis rather than by changing the underlying set representation. Our work is orthogonal: we keep the abstraction fixed and instead accelerate the sparse-set kernels shared across multiple analyses.

Parallel Pointer Analysis. Méndez-Lojo et al. [28] formulate Andersen's analysis in terms of graph rewiring rules. The algorithm is further implemented on GPUs [27], exploiting task-level parallelism rather than the instruction-level parallelism we target with SIMD. D4 [22] introduces an incremental, parallel analysis within each iteration of fixed-point computation. Nagaraj and Govindarajan [29] propose a flow-sensitive analysis based on the graph-rewriting rules in [28]. PSEGPT [45] is also a parallel flow-sensitive analysis but builds heap def-use chains on the fly. Su et al. [35] introduce a demand-driven, context-sensitive analysis that runs CFL-reachability operations in parallel and focuses on inter-query parallelism. Edvinsson et al. [8] present a GPU-based implementation of control flow analysis for functional languages. Zuo et al. [46] likewise target scalability for large systems code by systematizing interprocedural static analyses on a graph-processing substrate. Our approach complements these works by optimizing the fundamental set-theoretic operations on which they rely.

SIMD Optimizations for Data Structures. The use of SIMD instructions to accelerate fundamental data structures has attracted significant interest in recent years. Inoue et al. [17] demonstrate that SIMD-accelerated set intersection can significantly reduce branch mispredictions by processing multiple elements in parallel, directly motivating our approach to index matching. Lemire et al. [20] introduce Roaring Bitmaps, which use SIMD instructions to accelerate operations on compressed bitmaps. This work shares our focus on hardware-aware optimizations but targets different usage patterns and does not address the specific challenges of pointer analysis. Roaring organizes the universe into 2^{16}-sized containers and chooses among array/bitmap/run containers to compress within each key range. Its SIMD gains primarily come from container-local operations. For pointer-analysis workloads, however, its container structure is still organized around coarse universe partitions rather than the smaller, fixed block ranges that dominate inclusion-based set propagation. We include an explicit empirical comparison in § 5.

8 Conclusion

This paper presents a block-based sparse bit-vector representation for pointer analysis that uses SIMD vectorization to improve execution efficiency. As processor architectures continue to evolve, such hardware-aware designs are likely to play a central role in scaling static analyses to increasingly complex software systems.

Data Availability

The implementation and LLVM bitcodes used in evaluation are publicly available at: https://figshare.com/s/0c516d0aa7f871d1dea2

Acknowledgments. We sincerely thank the reviewers for their suggestions. This work is partially supported by the National Natural Science Foundation of China (62302434 and U2341212).

References

[1] Thomas Ball and Sriram K Rajamani. 2002. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. ACM, New York, NY, USA, 1–3. https://doi.org/10.1145/503272.503274

[2] Mohamad Barbar and Yulei Sui. 2021. Compacting Points-To Sets through Object Clustering. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/3485547

[3] Mohamad Barbar and Yulei Sui. 2021. Hash Consed Points-To Sets. In *Static Analysis Symposium (SAS)*. 25–48. https://doi.org/10.1007/978-3-030-88806-0_2

[4] Mohamad Barbar, Yulei Sui, and Shiping Chen. 2021. Object Versioning for Flow-Sensitive Pointer Analysis. (2021). https://doi.org/10.1109/CGO51591.2021.9370334

[5] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. 2003. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. 103–114. https://doi.org/10.1145/781131.781144

[6] Alessandro Colantonio and Roberto Di Pietro. 2010. Concise: Compressed 'n' Composable Integer Set. *Inform. Process. Lett.* 110, 16 (July 2010), 644–650. https://doi.org/10.1016/j.ipl.2010.05.018

[7] Guille Díez-Cañas. 2022. Faster-Than-Native Alternatives for x86 VP2INTERSECT Instructions. https://doi.org/10.48550/arXiv.2112.06342

[8] Marcus Edvinsson, Jonas Lundberg, and Welf Löwe. 2011. Parallel points-to analysis for multi-core machines. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers* (Heraklion, Greece) *(HiPEAC '11)*. ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/1944862.1944872

[9] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective Typestate Verification in the Presence of Aliasing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis* (Portland, Maine, USA) *(ISSTA '06)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/1146238.1146254

[10] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 9 (May 2008), 34 pages. https://doi.org/10.1145/1348250.1348255

[11] Yiyuan Guo, Peisen Yao, and Charles Zhang. 2024. Precise Compositional Buffer Overflow Detection via Heap Disjointness. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 63–75. https://doi.org/10.1145/3650212.3652110

[12] Ben Hardekopf and Calvin Lin. 2007. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In *Static Analysis*, Hanne Riis Nielson and Gilberto Filé (Eds.). Springer, Berlin, Heidelberg, 265–280. https://doi.org/10.1007/978-3-540-74061-2_17

[13] Benjamin Charles Hardekopf. 2009. *Pointer analysis: Building a foundation for effective program analysis*. Ph.D. https://www.proquest.com/docview/305002809/abstract/522F05B1FBCB47E6PQ/1

[14] Jiaqi He and Karim Ali. 2026. GPU-Accelerated Flow-Sensitive Pointer Analysis for C/C++ Programs. *Proceedings of the ACM on Software Engineering* 3, FSE (2026), 3116–3137. https://doi.org/10.1145/3808145

[15] Michael Hind. 2001. Pointer analysis: Haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, Utah, USA) *(PASTE '01)*. ACM, New York, NY, USA, 54–61. https://doi.org/10.1145/379605.379665

[16] Michael Hind and Anthony Pioli. 2000. Which pointer analysis should I use?. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Portland, Oregon, USA) *(ISSTA '00)*. ACM, New York, NY, USA, 113–123. https://doi.org/10.1145/347324.348916

[17] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. 2014. Faster Set Intersection with SIMD Instructions by Reducing Branch Mispredictions. In *Proceedings of the 40th International Conference on Very Large Data Bases*. 293–304. https://doi.org/10.14778/2735508.2735518

[18] ZJU Programming Languages and Automated Reasoning Group. 2025. Lotus: A Versatile and Industrial-Scale Program Analysis Framework. https://github.com/ZJU-PL/lotus Program analysis framework built on LLVM.

[19] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 278–289. https://doi.org/10.1145/1250734.1250766

[20] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. 2018. Roaring Bitmaps: Implementation of an Optimized Software Library. *Software: Practice and Experience* 48, 4 (April 2018), 867–895. https://doi.org/10.1002/spe.2560 arXiv:1709.07821 [cs].

[21] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 15:1–15:27. https://doi.org/10.4230/LIPIcs.ECOOP.2016.15

[22] Bozhen Liu and Jeff Huang. 2018. D4: fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 359–373. https://doi.org/10.1145/3192366.3192390

[23] Peiming Liu, Yanze Li, Brad Swain, and Jeff Huang. 2022. PUS: A Fast and Highly Efficient Solver for Inclusion-based Pointer Analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1781–1792. https://doi.org/10.1145/3510003.3510075 ISSN: 1558-1225.

[24] LLVM Project. 2026. LLVM: include/llvm/ADT/SparseBitVector.h Source File. https://llvm.org/doxygen/SparseBitVector_8h_source.html. Accessed: 2026-01-08.

[25] Magnus Madsen and Anders Møller. 2014. Sparse Dataflow Analysis with Pointers and Reachability. (2014), 241–261. https://doi.org/10.1007/978-3-319-10936-7_13

[26] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. (2012), 107–116. https://doi.org/10.1145/2145816.2145831

[27] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU Implementation of Inclusion-based Points-to Analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. ACM, New York, NY, USA, 107–116. https://doi.org/10.1145/2145816.2145831

[28] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. ACM, Reno/Tahoe Nevada USA, 428–443. https://doi.org/10.1145/1869459.1869495

[29] Vaivaswatha Nagaraj and R. Govindarajan. 2013. Parallel flow-sensitive pointer analysis by graph-rewriting. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques (PACT '13)*. IEEE Press, Edinburgh, Scotland, UK, 19–28. https://dl.acm.org/doi/10.5555/2523721.2523728

[30] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M.J. Harrold. 2004. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings. 26th International Conference on Software Engineering*. 491–500. https://doi.org/10.1109/ICSE.2004.1317471 ISSN: 0270-5257.

[31] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 693–706. https://doi.org/10.1145/3192366.3192418

[32] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-Sensitive Sparse Analysis without Path Conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 930–943. https://doi.org/10.1145/3453483.3454086

[33] Fabio Somenzi. 2012. CUDD: CU Decision Diagram Package. https://davidkebo.com/cudd/. Accessed: 2026-01-08.

[34] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 112–122. https://doi.org/10.1145/1250734.1250748

[35] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel Pointer Analysis with CFL-Reachability. In *Proceedings of the 2014 Brazilian Conference on Intelligent Systems (BRACIS '14)*. IEEE Computer Society, Washington, DC, USA, 451–460. https://doi.org/10.1109/ICPP.2014.54

[36] Yu Su, Ding Ye, Jingling Xue, and Xiang-Ke Liao. 2015. An efficient GPU implementation of inclusion-based pointer analysis. *IEEE Transactions on Parallel and Distributed Systems* 27, 2 (2015), 353–366. https://doi.org/10.1109/TPDS.2015.2397933

[37] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 254–264. https://doi.org/10.1145/2338965.2336784

[38] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486

[39] Chengpeng Wang, Wenyang Wang, Peisen Yao, Qingkai Shi, Jinguo Zhou, Xiao Xiao, and Charles Zhang. 2023. Anchor: Fast and Precise Value-Flow Analysis for Containers via Memory Orientation. *ACM Trans. Softw. Eng. Methodol.* 32, 3 (2023), 66:1–66:39. https://doi.org/10.1145/3565800

[40] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. (2005), 97–118. https://doi.org/10.1007/11575467_8

[41] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2006. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.* 31, 1 (March 2006), 1–38.

https://doi.org/10.1145/1132863.1132864

[42] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 327–337. https://doi.org/10.1145/3180155.3180178

[43] Peisen Yao, Zinan Gu, and Qingkai Shi. 2026. Phoenix: A Modular and Versatile Framework for C/C++ Pointer Analysis. In *Tool Demonstration Track, Joint Conference of ISSTA and SPLASH (SPLASH/ISSTA 2026)*.

[44] Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2024. Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis. In *Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 567–592. https://doi.org/10.1145/3656400

[45] Jisheng Zhao, Michael G Burke, and Vivek Sarkar. 2018. Parallel sparse flow-sensitive points-to analysis. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. ACM, New York, NY, USA, 59–70. https://doi.org/10.1145/3178372.3179517

[46] Zhiqiang Zuo, Kai Wang, Aftab Hussain, Ardalan Amiri Sani, Yiyu Zhang, Shenming Lu, Wensheng Dou, Linzhang Wang, Xuandong Li, Chenxi Wang, and Guoqing Harry Xu. 2020. Systemizing Interprocedural Static Analysis of Large-scale Systems Code with Graspan. *ACM Transactions on Computer Systems* 38, 1–2 (2020), 4:1–4:39. https://doi.org/10.1145/3466820

Received 2026-03-26; accepted 2026-06-18