Intelligent Multimedia Processing & Communication Systems Journal

IMPCSJ

J IMPCS (2025) 21: 59-66
DOI 10.71856/IMPCS.2025.1218351

Research Paper

A Time and Space-Efficient Compositional Method for Data Flow Analysis

Ebrahim Fazli*

Assistant Professor, Department of Computer Engineering, Za.C., Islamic Azad University, Zanjan, Iran.
***Corresponding Author*, efazli@iau.ac.ir**

Article Info

Article history:
Received: 30 Jun 2025
Accepted: 11 Aug 2025

Keywords:
Data flow analysis,
Fault detection,
Static program analysis.

ABSTRACT

This paper investigates the problem of static data flow analysis, which is an important problem in program comprehension, optimization, and bug detection. However, its application to large-scale, real-world software often faces significant challenges related to computational time and memory consumption, particularly for programs exhibiting extremely large Npath complexities. Existing approaches frequently struggle to scale efficiently while maintaining precision. To address this deficiency, we propose a novel compositional method for classical data flow analyses, specifically focusing on Available Expressions, Very Busy Expressions, Reaching Definitions, and Live Variables. Our approach leverages the decomposition of the Control Flow Graph into its Strongly Connected Components (SCCs), employing a divide-and-conquer strategy that analyzes smaller, more manageable program corpora. A key innovation is the Two-level Set Accessing Method (TSAM), a non-contiguous, pointer-based data structure that significantly reduces memory overhead for storing the dynamic data flow information. We prove that our algorithm, which utilizes a queueing mechanism for fixed-point computation, eventually terminates while achieving the same level of precision as traditional, exhaustive global analyses. Our extensive experimental evaluation against the traditional iterative method demonstrates that the SCC-based method, coupled with TSAM, significantly outperforms existing techniques, consuming on average 53% less memory and utilizing 43% less processing time, particularly for programs with high Npath complexity. This work provides a practical and scalable solution for precise data flow analysis of complex software systems.

I. Introduction

Program analysis plays a critical role in modern software engineering, enabling tasks ranging from compiler optimizations and performance tuning to automated vulnerability detection and software quality assurance. Static data flow analysis (DFA), in particular, provides a systematic way to gather information about the possible values of variables, the availability of expressions, or the liveness of data at different program points without executing the code. Classical data flow problems such as Available Expressions (AE), Very Busy Expressions (VBE), Reaching Definitions (RD), and Live Variables (LV) form the foundational bedrock for many advanced analyses.

Despite their fundamental importance, applying these classical DFA techniques to industrial-scale software projects presents formidable challenges. Modern codebases can contain millions of lines of code, intricate control flow, and complex interprocedural dependencies. A significant bottleneck arises from programs characterized by extremely large Npath complexities, where the sheer number of possible execution paths makes exhaustive analysis computationally intractable. Traditional iterative algorithms, while precise, often suffer from prohibitively high time and space complexity, rendering them impractical for large-scale applications. The need for scalable yet precise static analysis methods remains a pressing concern in the field.

This paper introduces a novel time and space-efficient compositional method for data flow analysis designed to overcome these scalability limitations. Our approach is grounded in the principle of divide-and-conquer, achieved through the decomposition of the program's Control Flow Graph (CFG) into its Strongly Connected Components (SCCs). By breaking down the analysis into smaller, self-contained SCCs and then composing their results, we can efficiently analyze complex program structures that would otherwise overwhelm conventional methods. This SCC-based strategy allows each phase of the analysis to work on a small corpus of the program under analysis, leading to significant practical benefits.

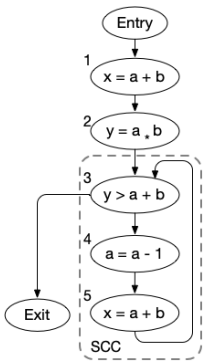
Contributions：The major contributions of this paper are multi-fold : First, we present a novel high-performance SCC-based compositional framework for performing AE, VBE, RD, and LV analyses by efficiently identifying, analyzing, and merging information across program SCCs. This approach ensures that each phase of the analysis operates on a significantly smaller corpus, leading to substantial performance gains. Second, we introduce Three-level Set Accessing Method (TSAM). It is a novel pointer-based, non-contiguous memory allocation data structure specifically designed to efficiently store and manage the dynamic sets of data flow information associated with program points. TSAM significantly mitigates the high space cost traditionally associated with storing abundant intermediate data flow facts, especially in large CFGs. Third, through extensive experimentation on a suite of benchmark programs, we demonstrate that our SCC-based compositional method, powered by TSAM, consistently outperforms the traditional iterative method in terms of both processing time and memory consumption. Our method achieves the same level of precision as traditional global analyses while demonstrating superior scalability for programs with extremely large Npath complexities. We show average reductions in memory consumption of 53% and processing time reductions of 64% compared to the iterative method.

Organization: Section II defines some basic concepts. Section III states the DFA problem. Subsequently, Section IV presents data structures as well as TSAM method of memory allocation. Section V offers a vertex-based algorithm for DFA generation. Section VI puts forward a highly time and space-efficient SCC-based method for DFA. Section VII presents our experimental results. Section VIII discusses related work. Finally, Section 8 makes concluding remarks and discusses future extensions of this work.

II. Preliminaries

This section presents some graph-theoretic concepts that we utilize throughout this paper. A directed graph $G = (V, A)$ includes a set of vertices V and a set of arcs $(v_i, v_j) \in E$, where $v_i, v_j \in V$. A vertex v_j is reachable from another vertex v_i iff (if and only if) there is a simple path that emanates from vi and terminates at vj. A SCC in G is a sub-graph $G' = (V',A')$, where $V' \subseteq V$ and $A' \subseteq A$, and for any pair of vertices vi,vj ∈ V′, vi and vj are reachable from each other. Tarjan [1] presents a polynomial-time algorithm that finds the SCCs of the input graph and constructs its component graph. Each vertex of the input graph appears in exactly one of the SCCs. The result is a Directed Acyclic Graph (DAG) whose every vertex is an SCC. A Control Flow Graph (CFG) models the flow of execution control between the basic blocks in a program, where a basic block is a collection of program statements without any conditional or unconditional jumps. Each vertex $v \in V$ corresponds to a basic block. Each edge/arc $e = (v_i, v_j) \in E$ corresponds to a possible transfer of control from block v_i to block v_j. A CFG often has a start vertex that captures the block of statement starting with the first instruction of the program, and has some end vertices representing the blocks of statements that end in a halt/exit/return instruction. (We use the terms ‘arc ’and ‘edge ’interchangeably throughout this paper.) Figure 1 illustrates an example method as well as its corresponding CFG for a class in the Apache Commons library.

```
Function Foo (a, b) {
1: var x = a + b;
2: var y = a * b;
3: while (y > a + b) {
4:       a = a + 1;
5:       x = x + b;
   } // while
} //Foo
```



(a) Source code example (b) CFG for function Foo
FIGURE 1. example function and corresponding CFG

Definition 1 (Component Graph of CFGs): The component graph of a CFG $G = (V, A)$, called CCFG, is a DAG whose vertices are the SCCs of G, and any arc $(v_i, v_j) \in A$ starts in an SCC_i and ends in SCC_j.

In the following definitions, let $G = (V, A)$ be a CFG and $C = (V_c, A_c)$ be an SCC in the CCFG of G; i.e., C is a vertex in CCFG of G.

Definition 2 (SccEntryVertex): A vertex $v_e \in V_c$ is an SccEntryVertex of C iff $\exists v : v \in V \wedge v \notin V_c : (v, v_e) \in A$. (e.g., Vertex 3 in Fig 1(b)).

Definition 3 (SccExitVertex): A vertex $v_e \in V_c$ is an SccExitVertex of C iff $\exists v : v \in V \wedge v \notin V_c : (v_e, v) \in A$. (e.g., Vertex 3 in Fig 1(b)).

III. Problem Statement

The primary goal of DFA is to determine how data flows through a program, enabling optimizations and the detection of potential errors. Static analysis of CFGs related to real world programs with a large Npath complexity is an important problem in compilers and program analysis tools to gather information about the possible states of a program at various points during its execution. To compute analysis state at each program point, DFA finds a fixed-point solution to a system of data flow equations derived from all components. For each vertex v, we seek to compute *v.entrySet* (information before v) and *v.exitSet* (information after v). DFA problem can be formulated as follows:

Problem 1 (DFA Generation):

Input:

1. A graph G = (V, E) that represents the CFG of a given program, a start vertex s∈V and an end vertex e∈V.
2. Dataflow Framework Definition: The formal specification of the particular data flow problem, comprising:
   a. Lattice of Dataflow Facts (L, ⊑, ⊓ / ⊔, ⊤, ⊥): The set of all possible data flow information at any program point, along with a partial order and a meet/join operator to combine information.
   b. Transfer Functions (fn): For each vertex v∈V, a function fn :L→L that describes how the data flow information are transformed when control passes through that vertex.
   c. Boundary Condition: The initial data flow information at the program's entry vertex (for forward problems) or exit vertex (for backward problems).

Output:

For each vertex v∈V in the input CFG, the desired output is a pair of data flow information sets:

1. v.entrySet (v_{IN}): The data flow information that are true immediately before the execution of vertex v.
2. v.exitSet(v_{OUT}): The data flow information that are true immediately after the execution of vertex v.

These sets must represent the Least Fixed Point (LFP) solution to the system of data flow equations defined by the problem's lattice, transfer functions, and boundary conditions.

IV. Data Structures

In this section, we present a data structure for storing the input CFG (Section 4.1), and a novel memory allocation method called TSAM (Section 4.2) for storing the generated DFAs.

IV.I. CFG Data Structure

A matrix is usually stored as a two-dimensional array in memory. Memory requirements of a sparse matrix can be significantly reduced by maintaining only non-zero entries. Based on the number and distribution of non-zero entries, we can use different data structures. The Compressed Sparse Row (CSR, CRS or Yale format) [8] represents a matrix by a one-dimensional array that supports efficient access and matrix operations. We employ the CSR data structure to maintain a directed graph, where vertices of the graph receive unique IDs in $\{0, 1, \cdot \cdot \cdot, |V| - 1\}$. To represent a graph in CSR format, we store end vertices and start vertices of arcs in two separate arrays EndV and StartV respectively (Figure 2). Each entry in EndV points to the starting index of its adjacency list in array StartV. For example, Figure 2 illustrates the CSR representation of the graph of Figure 1(b). Since the proposed algorithm computes all DFAs ending in each vertex v ∈ V, maintaining the predecessor vertices is of particular importance. In CSR data structure, first the vertex itself and then its predecessor vertices are stored.

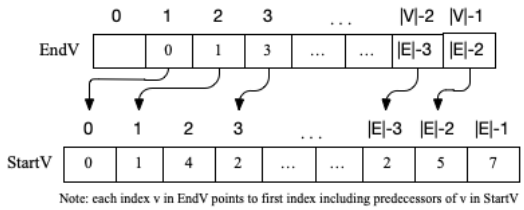


Fig. 2. CSR graph representation

IV.II. Two-level Set Accessing Method

A significant challenge in scaling data flow analysis to large programs is the immense memory footprint required to store the evolving sets of data flow information at every program point. For a large CFG, the number of such information sets (e.g., v_{IN} and v_{OUT} for every vertex v) could be enormous, which would incur a significant space cost on the algorithm. To reduce this space complexity, we introduce the Two-level Set Accessing Method (TSAM), a novel pointer-based, non-contiguous memory allocation data structure specifically designed for efficient storage and access of data flow sets.

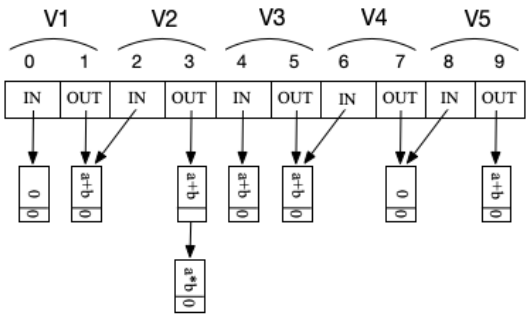


Fig. 3. Two-level Set Accessing Method (TSAM)

TSAM is an extension of the TPAM data structure from our previous work [12]. While TPAM was suitable for storing intermediate partial paths, TSAM has been specifically adapted and optimized to handle the dynamic and update-intensive nature of data flow sets (such as AE or LV) required by classical data flow analyses. This distinction is critical as data flow analysis involves frequent modifications, unions, and intersections of sets of facts, which differs fundamentally from the primarily sequential storage and retrieval of path segments.

TSAM is organized hierarchically into two levels, leveraging linked lists for flexibility and memory efficiency:

Level 1 (Address Table): This serves as the primary access mechanism. It is an address table (e.g., an array of pointers) with a length of 2|V| in the input CFG. For each vertex v , there are two pointers from Level 1 lead to its dedicated Level 2 structures.

Level 2 (Actual Set Contents): This level consists of two linked lists for each vertex (e.g., for its v_{IN} and v_{OUT}). This is where the actual data flow information sets reside. Each node within these lists represents a data flow fact (e.g., an expression, a variable, or a definition). For instance {a+b, a-b} is the AE at the entry point of statement 1, and {a+b} is the AE at exit point of statement 1(Figure 2).

TSAM achieves significant memory efficiency through several key mechanisms:

Non-Contiguous Allocation and Sparsity: By utilizing a pointer-based, non-contiguous memory allocation strategy TSAM only allocates memory precisely as needed for the actual data flow facts and their pointer-based organization. This inherently exploits the common sparsity observed in data flow sets, where many sets might be empty or contain only a few elements. Unlike fixed-size representations (e.g., large bitsets) that must pre-allocate space for the entire universe of possible facts for every program point, TSAM avoids this overhead by dynamically sizing its linked lists. If an Entry Set or ExitList is empty, its corresponding structure in Level 2 might be minimal or even a null pointer, further conserving memory.

Set Canonicalization/Sharing: The pointer-based nature of TSAM, which points to Level 2 content, creates a strong opportunity for set canonicalization (or hash-consing). If multiple vi EntryLists or vi ExitLists contain identical sets of information (e.g., the exact same set of "available expressions"), TSAM can store that set content once in Level 2. Multiple pointers from Level 1 can then reference this single canonical instance in Level 2, thereby avoiding redundant storage and significantly reducing overall memory consumption, especially when common data flow states recur across the CFG. This approach involves hashing set contents to identify duplicates and managing a pool of unique sets in Level 2.

Efficient Set Operations: TSAM needed to support rapid set union (∪), intersection (∩), and difference (−) operations. This was achieved by optimizing linked list traversals and, implicitly, by leveraging the canonicalization of sets in Level 2. When performing a set operation (e.g., S1 ∪ S2), the result Snew would be computed and then checked against the canonicalized sets in Level 2. If Snew already exists, a pointer to the existing instance is returned; otherwise, Snew is added to Level 2 and becomes a new canonical instance.

Dynamic Insertion and Deletion: Facts are not just accumulated; they are added and removed as information propagates and converges. The linked list implementation in Level 2 is inherently well-suited for dynamic insertion and deletion of individual facts compared to fixed-size arrays.

Dynamic Nature of Sets: The content of data flow sets changes frequently during the iterative fixed-point computation. TSAM's pointer-based approach allows for efficient updates: when a set's content changes, a new canonical set might be generated in Level 2, and the corresponding pointer in Level 2 is updated to point to this new set member. Old, unreferenced sets can then be garbage collected.

V. Vertex-Based Algorithm for DFA

This section presents a time-efficient algorithm (i.e., a solution for Problem 1) that takes a digraph G(V,A) (representing the CFG of a program) and a start vertex s∈V, and then computes two sets v_{IN} and v_{OUT} for each vertex v in the given CFG. Initially, the mentioned two sets are empty, and the start vertex s and its immediate successors are inserted in a queue Q. Algorithm 1 performs two kinds of processing on v_i: (1) computing data flow facts at the entry point of v_i (Lines 6 to 10), and (2) generating data flow facts at the exit point of v_i (Lines 12 to 14). Each vertex vi undergoes these processing steps after extraction from Q (Line 2). Algorithm 1 then propagates the wave of updates to the successors of v_i by inserting them in Q (Lines 15 to 17).

Algorithm 1 Vertex-Based DFA Generation // AE analysis
Input: *G*(*V* , *A*) with an outdegree 2; Start vertex *s* ∈ *V* ; *End vertex E*
Output: For each vertex v∈V in the input CFG, v.EntrySet, v.ExitSet.
Initialize: ∀v_i ∈ *V* , v_i.*IN* = v_i.*OUT* = empty, v_i.*updateFlag* = *false*, and queue *Q* with *s*.

1. **while (*Q* is non-empty) do**
2. Extract v_i from *Q*;
3. v_i.*updateFlag* = *false*;
4. **if** v_i =*s* **then**
5. Insert the immediate successors of v_i in *Q*;
6. **for each** v_j where (v_j, v_i) ∈ *A* **do**
7. **for each** entry *e* ∈ v_j.*OUT* **do**
8. **if** *e* is not read by v_i **then**
9. *Oj* = *Oj* ∪ *e*;
10. **for each** v_j where (v_j, v_i) ∈ *A* **do**
11. v_i.*INtemp* = ∩ *Oj*;

```
12.        if  vi.INtemp <> vi.IN then
13.              vi.IN = vi.INtemp;
14.              vi.OUT = vi.IN \ kill(vi) ∪ gen(vi) ;
15.              vi.updateFlag = true;
16.        if vi.updateFlag = true then
17.              for each vk where (vi,vk) ∈ A do
18.                    Insert vk in Q;
19. for each vi ∈V do
20.       return vi.IN;
21.       return vi.OUT;
```

Lemma 1: Let (v_j, v_i) be an arc in A and e be an entry in in $v_{j.OUT}$. The condition of the if statement on Line 8 evaluates to true for e and v_i no more than once. That is, each entry $e \in v_j.OUT$ is read by v_i at most once.

proof 1: Suppose Algorithm 1 has already entered the if statement on Line 8 for some e and v_i. This means that e has been labeled as read by v_i. Thus, next time Algorithm 1 gets to check if e is read by v_i, the condition on Line 8 evaluates to false.

Lemma 2: For any vertex v_j that has an outgoing arc (v_j, v_i), each entry $e \in v_j.OUT$ will eventually be labeled read by v_j.

proof 2: Initially, $v_j.OUT$ for each $v_j \in V$ is empty. The for-loop in Line 6 iterates through all incoming arcs of each vertex $v_i \in V$ and will eventually get to (v_i , v_i). As a result, any entry $e \in v_j.OUT$ will be labeled on Line 9 because initially all lists are empty. If new entry are imported in $v_j.OUT$ from its predecessors in subsequent iterations of the algorithm, then such entry will be labeled as read by v_i.

Lemma 3: For each vertex $v_i \in V$, at some finite point in time, $v_i.updateFlag$ will become false and will remain false.

proof 3: Lemmas 1 and 2 imply that all entries in the predecessors of v_i will eventually be labeled as read by v_i, and will keep their status of being read. As such, the condition on Line 8 will never become true again when processing arc (v_i, v_i). Therefore, Algorithm 1 will no longer get to Line 15; i.e., $v_i.updateFlag$ will become false (on Line 3) and will never become true again.

Theorem 1: Algorithm 1 will eventually terminate.

proof 4: Lemma 3 implies that at some finite point in time, the condition in Line 16 will become false for each $v_i \in V$ and will remain false. Thus, Algorithm 1 will eventually stop inserting vertices in Q. Moreover, the remaining vertices in Q are extracted on Line 2 in subsequent iterations of the while-loop. Thus, Q will eventually become empty; i.e., Algorithm 1 exits the while loop.

VI. Proposed Method

Our proposed compositional method for data flow analysis leverages a systematic decomposition of the input Control Flow Graph (CFG) into its Strongly Connected Components (SCCs), followed by a structured analysis and merging process. This divide-and-conquer paradigm is central to achieving high levels of time and space efficiency, particularly for programs with complex control flow structures and extremely large Npath complexities. For each vertex vi ∈V in the CFG, our analysis maintains two critical sets of information: vi.EntrySet to record all incoming data flow information and vi.ExitSet to record all outgoing information. Depending on the analysis direction (e.g., forwarding analysis for RD or AE), the ExitSet of a vertex is computed based on its EntrySet and the transfer function of the statement within that vertex. For reaching a fixed point and finalizing the analysis, the proposed method uses a queueing mechanism. We formally prove that this algorithm eventually terminates while guaranteeing the same level of precision as traditional, exhaustive global analyses.

The overall compositional analysis proceeds in three main phases:

Component Graph Computation: The initial step involves efficiently computing the component graph of the input CFG. This process transforms the CFG into a Directed Acyclic Graph (DAG) where each node represents an SCC. This decomposition is crucial for establishing an efficient processing order, as SCCs can be analyzed in topological order within the component DAG.

SCC-Local Information Calculation: For each SCC in the component graph, we calculate the sets of data flow information associated with its internal vertices. Since each SCC represents a smaller, self-contained portion of the program's control flow (often containing loops or recursion), traditional iterative data flow algorithms can be applied efficiently within these local contexts. The reduced size of the analysis domain within each SCC significantly improves performance.

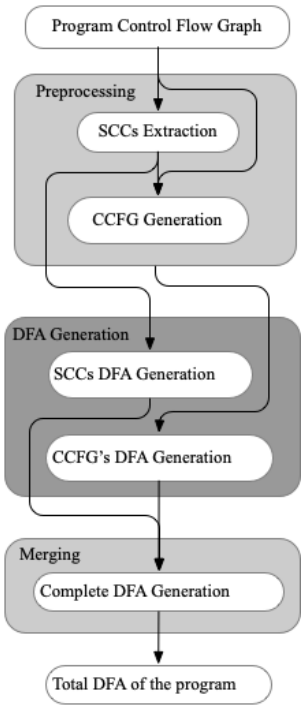


FIGURE 4. Overview of compositional method for DFA

Global Information Generation and Merging: Finally, the results from individual SCCs are systematically merged and propagated through the component graph to generate the final sets of data flow information for the entire CFG. This merging process respects the dependencies between SCCs, ensuring that global precision is maintained and that information flows correctly across SCC boundaries.

VII. Experimental Results

This section presents the results of our experimental evaluations of the proposed SCC-based method for DFA compared to the iterative Vertex-based approach. To validate the effectiveness and efficiency of our proposed SCC-based compositional method, we implemented our approach and conducted a comprehensive experimental evaluation. We compared its performance against the traditional iterative method, which represents a common state-of-the-art baseline for precision in static data flow analysis.

TABLE 1. Modified benchmark CFGs

CFG	Original Functions	Graph structure after modification						
		Nodes	Edges	SCC	SccNodes	SccEdges	CC	Npath
1	AsmClassReaderAccept	180	214	18	78	83	35	2.1e7
2	AsmClassWriterToByteArray	215	258	24	103	110	44	6.1e11
3	SquareMesh2DcreateLinks	244	290	27	115	125	49	3.3e12
4	PrivilizerAsmMethodWriter	355	431	38	160	173	68	4.5e22
5	SingularValueDecomposition	486	567	47	223	244	104	1.1e23
6	ListParserTokenManager	723	853	75	331	351	131	2.0e32
7	BOBYQAOptimizer	874	994	83	409	762	155	9.3e39
8	ParserParserTokenManager	963	1119	93	448	490	213	1.3e44
9	InternalXsltcCompilerCUP	1441	1713	149	626	712	273	4.1e68
10	XPathLexerNextToken	2160	2566	224	957	1073	404	8.4e97

The experimental benchmark consists of a set of ten modified CFGs from [13] (which are taken from Apache Commons libraries. Table 1 presents the structure of these CFGs. Columns 3 to 9 of Table 2 provide the number of nodes, edges, and SCCs of each CFG. The total numbers of nodes and edges of all SCCs are mentioned as SccNodes and SccEdges, respectively. Columns 7 and 8 show the Cyclomatic Complexity (CC) [14] and Npath Complexity [15] of the input CFGs. The last column illustrates the number of prime paths produced with the GPU-based method. We compare the SCC-based and the Vertex-based approaches with respect to their running time. We ran all the experiments on an Intel Core i7 machine with 3.6GHz X 8 processors and 16 GB of memory running Ubuntu 17.01 with gcc version 5.4.1.

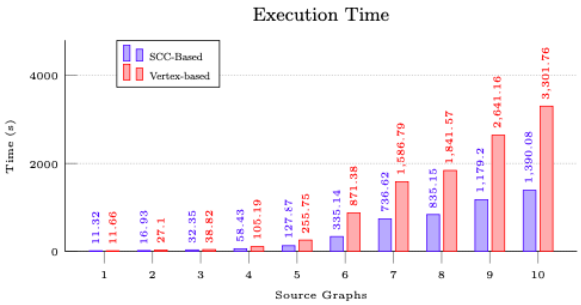


FIGURE 5. Time costs of the vertex-based and SCC-based algorithms on the benchmarks CFGs.

The bar graph of Figure 5 illustrates the time efficiencies of the SCC-based and Vertex-based approaches. (The reported timings for each approach is the average of fifteen runs.) These values reflect the fact that the time the SCC-based method is more effective for larger CFGs. Specifically, for the CFGs of the bottom four rows of Table 1. On average, the SCC-based method consumed 43% less time than the iterative Vertex-based method. This time efficiency increases significantly with growing graph size. For example, the SCC-based time efficiency in the last graph is 65%. The recorded times indicate that by increasing the structural complexity, the SCC-based algorithm provides a better performance. Thus, for real-world applications that have a large number of lines and complex structures, the SCC-based algorithm is expected to be highly efficient.

VIII. Related Works

This section discusses related works on the DFA. Static DFA has been an active area of research for decades, with numerous iterative and compositional approaches developed to address the inherent challenges of program complexity and scalability. The theoretical foundations for DFA were largely established by Kildall's seminal work, which introduced the lattice-theoretic framework and the traditional iterative method for computing fixed points [2]. This iterative approach, which repeatedly propagates data flow information until convergence over the entire CFG, serves as a common baseline for precision for classical problems such as RD, LV, AE, and VBE [3].

However, the scalability of the traditional iterative method often becomes a significant bottleneck for large and complex real-world programs [5]. Factors contributing to this include the number of iterations required for convergence, particularly in the presence of complex loops or irreducible CFGs, and the substantial memory footprint needed to store data flow facts for every program point [4]. Modern programs with intricate control flow structures and high Npath complexities can lead to an enormous number of program points and potential data flow facts, making brute-force iterative propagation impractical. Researchers have continuously sought methods to improve the efficiency of iterative algorithms, for instance, by optimizing iteration orders (e.g., reverse postorder for forward problems) or refining convergence tests [3, 7].

To overcome the inherent limitations of global iterative approaches, compositional data flow analysis methods have been extensively explored. These techniques aim to analyze program components independently and then combine their results, thereby reducing the scope of analysis at any given time. Early compositional methods often focused on modular analysis of functions or procedures. More advanced techniques have explored various program decomposition strategies, such as interval analysis, which partitions the CFG into single-entry regions, or the use of program dependence graphs (PDGs) and code property graphs (CPGs) that integrate control and data dependencies for comprehensive code representation [8, 9].

Our work builds upon the concept of decomposing the CFG into SCCs. SCC-based decomposition has been recognized as a powerful technique for handling loops and recursive structures efficiently in graph algorithms, allowing for a topological ordering of components that can simplify fixed-point computation. While SCCs have been used in various program analysis contexts, their specific application to the precise and efficient computation of classical DFA problems, while guaranteeing full precision, remains an area

of ongoing research, particularly when combined with a novel memory management technique.

Furthermore, the memory efficiency of data flow analysis is heavily reliant on the underlying data structures used to represent and manipulate sets of facts. Common representations include bitsets, sparse sets, hash-based collections, and various forms of pointer-based data structures designed for dynamic data [9]. Recent advancements in scalable static analysis also explore demand-driven approaches, which compute information only when needed to save both time and memory, particularly for complex analyses like points-to analysis [10]. Approaches for binary analysis also leverage specialized graph representations and optimized data flow analysis to handle large firmware images efficiently [11].

Our proposed Two-level Set Accessing Method (TSAM) extends our previous work on memory-efficient data structures. Specifically, TSAM builds upon the TPAM (Three-level Pointer Accessing Method) data structure introduced in our earlier work [12]. A key distinction and limitation of TPAM was its primary suitability for storing intermediate partial paths, which involves a relatively stable set of facts (path segments) that are primarily accumulated for enumeration. In contrast, data flow analysis requires storing dynamically changing sets of propagated information, such as AE or LV. This involves frequent and complex set operations (union, intersection, difference) and dynamic insertions/deletions of facts during fixed-point iteration. TSAM has been specifically adapted and optimized to handle this dynamic and update-intensive nature, aiming for superior memory efficiency for storing and accessing facts during iterative fixed-point computations in the context of DFA.

IX. Conclusion and Future Work

This paper presented a novel time and space-efficient compositional method for Data Flow Analysis (DFA), specifically targeting Available Expressions (AE), Very Busy Expressions (VBE), Reaching Definitions (RD), and Live Variables (LV). By decomposing Control Flow Graphs (CFGs) into Strongly Connected Components (SCCs) and employing a rigorous divide-and-conquer strategy, our method effectively addresses the scalability challenges posed by large-scale, real-world programs with high Npath complexities. A cornerstone of our approach is the Two-level Set Accessing Method (TSAM), a specialized pointer-based data structure that drastically reduces the memory footprint associated with storing dynamic data flow information. We have formally proven the termination of our fixed-point algorithm and demonstrated that our method achieves the same high level of precision as traditional, exhaustive global analyses.

The proposed compositional method (i) computes the component graph of the input CFG; (ii) calculates the set of facts of each SCC in the component graph, and (iii) generates the set of facts of the given CFG with very low time and space costs. We implemented and evaluated the proposed methods versus existing approaches, and our experimental results show that the proposed methods significantly outperform the state-of-the-art in dealing with programs that have extremely large Npath complexities (see Figure 5).

Building upon the robust framework presented in this paper, several promising avenues for future research exist:

Extension to Other Data Flow Problems: Investigate the applicability and benefits of our SCC-based compositional method and TSAM to a broader range of data flow problems, including more complex analyses such as constant propagation, taint analysis, and points-to analysis.

Inter-procedural Analysis: Extend the current intraprocedural framework to a full interprocedural analysis. This would involve adapting the SCC decomposition and information merging to handle function calls, returns, and contextual sensitivities.

Parallel Implementation: Explicitly implement and evaluate the parallelization strategies afforded by the SCC-based decomposition. Analyzing independent SCCs concurrently on multi-core processors (GPU-based) or distributed systems could yield even greater speedups.

Application to Diverse Codebases: Apply the method to an even wider range of larger, more diverse real-world open-source projects or industrial codebases to further validate its robustness and scalability in varied programming language and application contexts.

TSAM Optimization: Further optimize the TSAM data structure, potentially exploring hybrid representations (e.g., linked lists for sparse sets, bitsets for dense sets) or alternative canonicalization techniques to maximize memory efficiency and access speed.

Formal Benchmarking for Npath Complexity: Develop or leverage standardized benchmark suites specifically designed to evaluate static analysis tools on programs with extremely high Npath complexities, providing a more direct comparison metric for scalability in such challenging scenarios.

REFERENCES

[1] Tarjan, Robert. "Depth-first search and linear grajh algorithms." 12th Annual Symposium on Switching and Automata Theory (swat 1971). IEEE Computer Society, 1971.

[2] Kildall, Gary A. "A unified approach to global program optimization." Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1973.

[3] Muchnick, Steven. Advanced compiler design implementation. Morgan kaufmann, 1997.

[4] Cooper, Keith D., Timothy J. Harvey, and Ken Kennedy. "Iterative data-flow analysis, revisited." Rice Univ., Houston, TX, Tech. Rep. TR04-100, 2004.

[5] Hecht, Matthew S. Flow analysis of computer programs. Elsevier Science Inc., 1977.

[6] Nielson, Flemming, Hanne R. Nielson, and Chris Hankin. Principles of program analysis. Springer Science & Business Media, 2004.

[7] Yamaguchi, Fabian, et al. "Modeling and discovering vulnerabilities with code property graphs." 2014 IEEE symposium on security and privacy. IEEE, 2014.

[8] Akinyemi, Temidayo, et al. "A Comprehensive Review of Static Memory Analysis." IEEE Access (2024).
[9] Eisenstat, Stanley C., Martin H. Schultz, and Andrew H. Sherman. "Algorithms and data structures for sparse symmetric Gaussian elimination." SIAM Journal on Scientific and Statistical Computing 2.2, 1981.
[10] Yang, Xuqing. "Boosting static bug detection via demand-driven points-to analysis." Third International Conference on Communications, Information System, and Data Science (CISDS 2024). Vol. 13519. SPIE, 2025.
[11] Gibbs, Wil, et al. "Operation mango: Scalable discovery of Taint-Style vulnerabilities in binary firmware services." 33rd USENIX Security Symposium (USENIX Security 24). 2024.
[12] Fazli, Ebrahim, and Ali Ebnenasir. "TPGen: A Self-stabilizing GPU-Based Method for Test and Prime Paths Generation." International Conference on Fundamentals of Software Engineering. Cham: Springer Nature Switzerland, 2023.
[13] Bang, Lucas, Abdulbaki Aydin, and Tevfik Bultan. "Automatically computing path complexity of programs." Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015.
[14] McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering (4), 1976.
[15] Nejmeh, B.A.: Npath: a measure of execution path complexity and its applications. Communications of the ACM 31(2), 1988.