Efficient Interprocedural Data-Flow Analysis using Treedepth and Treewidth

Amir Kafshdar Goharshady and Ahmed Khaled Zaher

The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
goharshady@cse.ust.hk, akazaher@cse.ust.hk

Abstract. We consider interprocedural data-flow analysis as formalized by the standard IFDS framework, which can express many widely-used static analyses such as reaching definitions, live variables, and null-pointer. We focus on the well-studied on-demand setting in which queries arrive one-by-one in a stream and each query should be answered as fast as possible. While the classical IFDS algorithm provides a polynomial-time solution for this problem, it is not scalable in practice. More specifically, it will either require a quadratic-time preprocessing phase or takes linear time per query, both of which are untenable for modern huge codebases with hundreds of thousands of lines. Previous works have already shown that parameterizing the problem by the treewidth of the program's control-flow graph is promising and can lead to significant gains in efficiency. Unfortunately, these results were only applicable to the limited special case of same-context queries.

In this work, we obtain significant speedups for the general case of on-demand IFDS with queries that are not necessarily same-context. This is achieved by exploiting a new graph sparsity parameter, namely the treedepth of the program's call graph. Our approach is the first to exploit the sparsity of control-flow graphs and call graphs at the same time and parameterize by both the treewidth and the treedepth. We obtain an algorithm with a linear preprocessing phase that can answer each query in constant time wrt the size of the input. Finally, our experimental results demonstrate that our approach outperforms the classical IFDS and its on-demand variant by several orders of magnitude.

Keywords: Static Analysis · Data-flow Analysis · IFDS · Parameterized Algorithms.

1 Introduction

Data-flow. Data-flow analysis is a catch-all term for a wide and expressive variety of static program analyses that include common tasks such as reaching definitions [20], points-to and alias analysis [58,60,61,64,65,66], null-pointer dereferencing [43,40,22], uninitialized variables [46] and dead code elimination [29], as well as several other standard frameworks, e.g. gen-kill and bit-vector problems [35,36,33]. The common thread among data-flow analyses is that they consider certain "data facts" at each line of the code and then try to ascertain which data facts hold at any given point [53]. This is often achieved by a worklist algorithm that keeps discovering new data facts until it reaches a fixed point

and converges to the final solution [53,34]. Variants of data-flow analysis are already included in most IDEs and compilers. For example, Eclipse has support for various data-flow analyses, such as unused variables and dead code elimination, both natively [25] and through plugins [47,21]. Data-flow analyses have also been applied in the context of compiler optimization, e.g. for register allocation [38] and constant propagation analysis [28,57,14]. Additionally, they have found important use-cases in security [15], including in taint analysis [2] and detection of SQL injection attacks [27]. Due to their apparent importance, data-flow analyses have been widely studied by the verification, compilers, security and programming languages communities over the past five decades and are also included in program analysis frameworks such as Soot [6] and WALA [1].

Intraprocedural vs Interprocedural Analysis. Traditionally, data-flow analyses are divided into two general groups [32]:

- *Intraprocedural* approaches analyze each function/procedure of the code in isolation [33,21]. This enables modularity and helps with efficiency, but the tradeoff is that the call-context and interactions between the different procedures are not accounted for, hence leading to relatively lower precision.
- In contrast, *interprocedural* analyses consider the entirety of the program, i.e. all the procedures, at the same time. They are often sensitive to call context and only focus on execution paths that respect function invocation and return rules, i.e. when a function ends, control has to return to the correct site of the last call to that function [53,19]. Unsurprisingly, interprocedural analyses are much more accurate but also have higher complexity than their intraprocedural counterparts [50,53,56,59].

IFDS. One of the most classical and widely-used frameworks for interprocedural data-flow analysis is that of *Interprocedural Finite Distributive Subset problems* (IFDS) [53,52]. IFDS is an expressive framework that can perform all the analyses enumerated above by assigning a set D of data facts to each line of the program and then applying a reduction to a variant of graph reachability with side conditions ensuring that function call and return rules are enforced. For example, in a null-pointer analysis, each data fact d_i in D is of the form "the pointer p_i might be null". See Section 2 for details. Given a program with n lines, the original IFDS algorithm in [53] solves the data-flow problem *for a fixed starting point* in time $O(n \cdot |D|^3)$. Due to its elegance and generality, this framework has been thoroughly studied by the community. It has been extended to various platforms and settings [2,42,7], notably the on-demand setting [31] and in presence of correlated method calls [49], and has been implemented in standard static analysis tools [1,6].

On-demand Data-flow Analysis. Due to the expensiveness of exhaustive data-flow analysis, i.e. an analysis that considers every possible starting point, many works in the literature have turned their focus to on-demand analysis [31,17,3,61,65,66,24,51]. In this setting, the algorithm can first run a preprocessing phase in which it collects some information about the program and produces summaries that can be used to speedup the query phase. Then, in the query phase, the algorithm is provided with a series of queries and should answer each one as efficiently as possible. Each query is of the form $(\ell_1, d_1, \ell_2, d_2)$

and asks whether it is possible to reach line ℓ_2 of the program, with the data fact d_2 holding at that line, assuming that we are currently at line ℓ_1 and data fact d_1 holds[1]. It is also noteworthy that on-demand algorithms commonly use information found in previous queries to handle the current query more efficiently. On-demand analyses are especially important in just-in-time compilers and their speculative optimizations [17,18,39,4,26], in which having dynamic information about the current state of the program can dramatically decrease the overhead for the compiler. In addition, on-demand analyses have the following merits (quoted from[31,52]):

- narrowing down the focus to specific points of interest,
- narrowing down the focus to specific data-flow facts of interest,
- reducing the work in preliminary phases,
- side-stepping incremental updating problems, and
- offering on-demand analysis as a user-level operation that helps programmers with debugging.

On-demand IFDS. An on-demand variant of the IFDS algorithm was first provided in [31]. This method has no preprocessing but memoizes the information obtained in each query to help answer future queries more efficiently. It outperforms the classical IFDS algorithm of [53] in practice, but the only theoretical guarantee is that of same worst-case complexity, i.e. the on-demand version will never be any worse than running a new instance of the IFDS algorithm for each query. Hence, the worst-case runtime on m queries is $O(n \cdot m \cdot |D|^3)$. Recall that n is the number of lines in the program and $|D|$ is the number of data facts at each line. Alternatively, one can push all the complexity to the preprocessing phase, running the IFDS algorithm exhaustively for each possible starting point, and then answering queries by a simple table lookup. In this case, the preprocessing will take $O(n^2 \cdot |D|^3)$. Unfortunately, none of these two variants are scalable enough to handle codebases with hundreds of thousands of lines, e.g. standard utilities in the DaCapo benchmark suite [5] such as Eclipse or Jython.

Same-context On-demand IFDS. The work [17] provides a parameterized algorithm for a special case of the on-demand IFDS problem. The main idea in [17] is to observe that control-flow graphs of real-world programs are sparse and tree-like and that this sparsity can be exploited to find faster algorithms for *same-context* IFDS analysis. More specifically, the sparsity is formalized by a graph parameter called treewidth [55,54]. Intuitively speaking, treewidth is a measure of how much a given graph resembles a tree, i.e. more tree-like graphs have smaller treewidth. See Section 3 for a formal definition. It is proven that structured programs in several languages, such as C, have bounded treewidth [62] and there are experimental works that establish small bounds on the treewidth of control-flow graphs of real-world programs written in other languages, such as Java [30], Ada [13] and Solidity [16]. Using these facts, [17] provides an on-demand algorithm with $O(n \cdot |D|^3)$ preprocessing time and $O\left(\lceil \frac{|D|}{\lg n} \rceil\right)$ time per

[1] Instead of single data facts d_1 and d_2, we can also use a set of data facts at each of ℓ_1 and ℓ_2, but as we will see in Section 2, this does not affect the generality.

query. In practice, $|D|$ is often tiny in comparison with n and hence this algorithm is considered to have linear preprocessing and constant query time. Unfortunately, the algorithm in [17] is not applicable to the general case of IFDS and can only handle *same-context* queries. Specifically, the queries in [17] provide a tuple $(\ell_1, d_1, \ell_2, d_2)$ just as in standard IFDS queries but they ask whether it is possible to reach (ℓ_2, d_2) from (ℓ_1, d_1) by an execution path that *preserves the state of the stack*, i.e. ℓ_1 and ℓ_2 are limited to being in the same function and the algorithm only considers execution paths in which every function call returns before reaching ℓ_2.

Our Contribution. In this work, we present a novel algorithm for the general case of on-demand IFDS analysis. Our contributions are as follows:

- We identify a new sparsity parameter, namely the treedepth of the program's call graph, and use it to find more efficient parameterized algorithms for IFDS data-flow analysis. Hence, our approach exploits the sparsity of both call graphs and control-flow graphs and bounds both the treedepth and the treewidth. Treedepth [45,11] is a well-studied graph sparsity parameter. It intuitively measures how much the graph resembles a star, i.e. a shallow tree [44, Chapter 6].
- We provide a scalable algorithm that is not limited to same-context queries as in [17] and is much more efficient than the classical on-demand IFDS algorithm of [31]. Specifically, after a lightweight preprocessing that takes $O(n \cdot |D|^3 \cdot \text{treedepth}^2)$ time, our algorithm is able to answer each query in $O(|D|^3 \cdot \text{treedepth})$. Thus, this is the first algorithm that can solve the general case of on-demand IFDS scalably and handle codebases and programs with hundreds of thousands or even millions of lines of code.
- Our algorithm is perfectly parallelizable, in both preprocessing and querying. Hence, significant further modularity and scalability can be obtained by using several processors in parallel.
- We provide experimental results on the standard DaCapo benchmarks [5] illustrating that:
  - our assumption of the sparsity of call graphs and low treedepth holds in practice in real-world programs; and
  - our approach comfortably beats the runtimes of exhaustive and on-demand IFDS algorithms [53,31] by several orders of magnitude.

Novelty. Our approach is novel in several directions:

- Unlike previous optimizations for IFDS that only focused on control-flow graphs, we exploit the sparsity of both control-flow graphs and call graphs.
- To the best of our knowledge, this is the first time that the treedepth parameter is exploited in a static analysis or program verification setting. While this parameter is well-known in the graph theory community and we argue that it is a natural candidate for formalizing the sparsity of call graphs (See Section 3), this is the first work that considers it in this context.
- We provide the first theoretical improvements in the runtime of general on-demand data-flow analysis since [31], which was published in 1995.

– Our algorithm is much faster than [31] in practice and is the first to enable on-demand interprocedural data-flow analysis for programs with hundreds of thousands or even millions of lines of code. Previously, for such large programs, the only choices were to either apply the data-flow analysis intraprocedurally, which would significantly decrease the precision, or to limit ourselves to the very special case of same-context queries [17].

Limitation. The primary limitation of our algorithm is that it relies on the assumption of bounded treewidth for control-flow graphs and bounded treedepth for call graphs. In both cases, it is theoretically possible to generate pathological programs that have arbitrarily large width/depth: [30] shows that it is possible to write Java programs whose control-flow graphs have any arbitrary treewidth. However, such programs are highly unrealistic, e.g. they require a huge number of labeled nested while loops with a large nesting depth and break/continue statements that reference a while loop that is many levels above in the nesting order. Similarly, in Section 3, we construct a pathological example program whose call graph has a large treedepth. Nevertheless, this is also unrealistic and real-world programs, such as those in the DaCapo benchmark suite, have both small treewidth and small treedepth, as shown in Section 5 and [62,30,13,16].

Organization. In Section 2, we present the standard IFDS framework and formally define our problem. This is followed by a presentation of the graph sparsity parameters we will use, i.e. treewidth and treedepth, in Section 3. Our algorithm is then presented in Section 4, followed by experimental results in Section 5.

2 The IFDS Framework

In this section, we provide an overview of the IFDS framework following the notation and presentation of [17,53] and formally define the interprocedural data-flow problem considered in this work.

Model of Computation. Throughout this paper, we consider the standard word RAM model of computation in which every word is of length $w = \Theta(\lg n)$, where n is the length of the input. We assume that common operations, such as addition, shift and bitwise logic between a pair of words, take $O(1)$ time. Note that this has no effect on the implementation of our algorithms since most modern computers have a word size of at least 64 and we are not aware of any possible real-world input to our problems whose size can potentially exceed 2^{64}.

Control-flow Graphs. In IFDS, a program with k functions $f_1, f_2, \ldots, f_k$ is modeled by k control-flow graphs $G_1, G_2, \ldots, G_k$, one for each function, as well as certain interprocedural edges that model function calls and returns. The graphs G_i are standard control-flow graphs, having a dedicated *start vertex* s_i modeling the beginning point of f_i, another dedicated *end vertex* e_i modeling its end point, one vertex for every line of code in f_i, and a directed edge from u to v, if line v can potentially be reached right after line u in some execution of the program. The only exception is that function call statements are modeled by two vertices: a *call* vertex c_l and a *return site* vertex r_l. The vertex c_l has only incoming edges,

whereas r_l has only outgoing edges. There is also an edge from c_l to r_l, which is called a *call-to-return-site* edge. This edge is used to pass local information, e.g. information about the variables in f_i that are unaffected by the function call, from c_l to r_l.

Supergraphs. The entire program is modeled by a *supergraph* G, consisting of all the control-flow graphs G_i, as well as interprocedural edges between them. If a function call statement in f_i, corresponding to vertices c_l and r_l in G_i, calls the function f_j, then the supergraph contains the following interprocedural edges:

- a *call-to-start* edge from the call vertex c_l to the start vertex s_j of the called function f_j, and
- an *exit-to-return-site* edge from the endpoint e_j of the called function f_j back to the return site r_l.

Call Graphs. Given a supergraph G as above, a call graph is a directed graph C whose vertices are the functions $f_1, \ldots, f_k$ of the program and there is an edge from f_i to f_j iff there is a function call statement in f_i that calls f_j. In other words, the call graph models the interprocedural edges in the supergraph and the supergraph can be seen as a combination of the control-flow and call graphs.

Example. Figure 1 shows a program consisting of two functions (left) and its supergraph (right).

```
1  void g(int *&a, int *&b) {
2      b = a;
3  }
4
5  int main() {
6      int *a, *b;
7      a = new int(42);
8      g(a, b);
9      *b = 0;
10 }
```

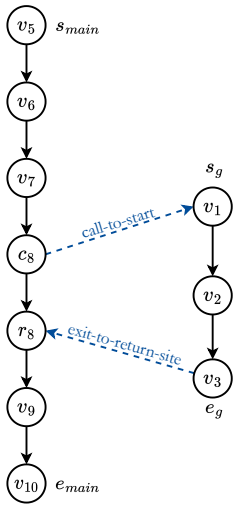

Fig. 1. A program (left) and its supergraph (right).

Valid Paths. The supergraph G potentially contains invalid paths, i.e. paths that are not realizable by an actual run of the underlying program. The IFDS framework only considers *interprocedurally valid* paths in G. These are the paths

that respect the rules for function invocation and return. More concretely, when a function's execution ends, control should return to the correct return-site vertex in its parent function. Formally, consider a path Π in G and let Π^* be the subsequence of Π that is obtained by removing any vertex that was not a call vertex c_l or a return-site vertex r_l. Then, Π is called a *same-context interprocedurally valid* path if Π^* can be generated from the non-terminal S in the following grammar:

$$S \rightarrow \quad \epsilon \quad | \quad c_l \quad S \quad r_l \quad S.$$

In other words, any function call in Π that was invoked in line c_l should end by returning to its corresponding return-site r_l. A same-context valid path preserves the state of the function call stack. In contrast, the path Π is *interprocedurally valid* or simply *valid* if Π^* is generated by the non-terminal S' in the following grammar:

$$S' \rightarrow \quad S \quad | \quad S' \quad c_l \quad S.$$

A valid path has to respect the rules for returning to the right return-site vertex after the end of each function, but it does not necessarily keep the function call stack intact and it is allowed ot have function calls that do not necessarily end by the end of the path. Let u and v be vertices in the supergraph G. We denote the set of all same-context valid paths from u to v by $\texttt{SCVP}(u, v)$ and the set of all interprocedurally valid paths from u to v by $\texttt{IVP}(u, v)$. In IFDS, we only focus on valid paths and hence the problem is to compute a *meet-over-all-valid-paths* solution to data-flow facts, instead of the *meet-over-all-paths* approach that is usually taken in intraprocedural data-flow analysis [53].

IFDS Arena [53]. An *arena* of the IFDS data-flow analysis is a tuple $(G, D, \Phi, M, \sqcap)$ wherein:

- $G = (V, E)$ is a supergraph consisting of control-flow graphs and interprocedural edges, as illustrated above.
- D is a finite set of *data facts*. Intuitively, we would like to keep track of which subset of data facts in D hold at any vertex of G (line of the program).
- The *meet operator* $\sqcap$ is either union or intersection, i.e. $\sqcap \in \{\cup, \cap\}$.
- Φ is the set of *distributive flow functions* over $\sqcap$. Every function $\varphi \in \Phi$ is of the form $\varphi : 2^D \rightarrow 2^D$ and for every pair of subsets of data facts $D_1, D_2 \subseteq D$, we have $\varphi(D_1 \sqcap D_2) = \varphi(D_1) \sqcap \varphi(D_2)$.
- $M : E \rightarrow \Phi$ is a function that assigns a distributive flow function to every edge of the supergraph. Informally, $M(e)$ models the effect of executing the edge e on the set of data facts. If the data facts that held before the execution of the edge e are given by a subset $D' \subseteq D$, then the data facts that hold after e are $M(e)(D') \subseteq D$.

We can extend the function M to any path Π in G. Let Π be a path consisting of the edges $e_1, e_2, \ldots, e_\pi$. We define $M(\Pi) := M(e_\pi) \circ M(e_{\pi-1}) \circ \cdots \circ M(e_1)$. Here, $\circ$ denotes function composition. According to this definition, $M(\Pi)$ models the effect that Π's execution has on the set of data facts that hold in the program's current state.

Problem Formalization. Consider an initial state $(u, D_1) \in V \times 2^D$ of the program, i.e. we are at line u of the program and we know that the data facts

in D_1 hold. Let $v \in V$ be another line, we define

$$\texttt{MIVP}(u, D_1, v) \ := \bigsqcap_{\Pi \in \texttt{IVP}(u,v)} M(\Pi)(D_1).$$

We simplify the notation to $\texttt{MIVP}(v)$, when the initial state is clear from the context. Our goal is to compute the MIVP values. Intuitively, MIVP corresponds to *meet-over-all-valid-paths.* If the meet operator is intersection, then $\texttt{MIVP}(v)$ models the data facts that *must* hold whenever we reach v. Conversely, if we use union as our meet operator, then $\texttt{MIVP}(v)$ is the set of data facts that *may* hold when reaching v. The work [53] provides an algorithm that for computing $\texttt{MIVP}(v)$ for every end vertex v in $O(n \cdot |D|^3)$, in which $n = |V|$.

Same-context IFDS. We can also define a same-context variant of MIVP as follows:

$$\texttt{MSCVP}(v) \ := \bigsqcap_{\Pi \in \texttt{SCVP}(u,v)} M(\Pi)(D_1).$$

The intuition is similar to MIVP, except that in MSCVP we only consider same-context valid paths that preserve the function call stack's status and ignore other valid paths. The work [17] uses parameterization by treewidth of the control-flow graphs to obtain faster algorithms for computing MSCVP. However, its algorithms are limited to the same-context setting. In contrast, in this work, we follow the original IFDS formulation of [53] and focus on MIVP, not MSCVP. Our main contribution is that we present the first theoretical improvement for computing MIVP since [53,31].

Dualization. In this work, we only consider the cases in which the meet operator is union. In other words, we focus on *may* analyses. IFDS instances in which the meet operator is intersection, also known as *must* analyses, can be reduced to union instances by a simple dualization. See [53] for details.

Data Fact Domain. In our presentation, we are assuming that there is a fixed global data fact domain D. In practice, the domain D can differ in every function of the program. For example, in a null-pointer analysis, the data facts in each function keep track of nullness of the pointers that are either global or local to that particular function. However, having different D sets would reduce the elegance of the presentation and has no real effect on any of the algorithms. So, we follow [53,17] and consider a single domain D in the sequel. Our implementation in Section 5 supports different domains for each function.

Graph Representation of Functions [53]. Every union-distributive function $\varphi : 2^D \to 2^D$ can be succinctly represented by the following relation $R_\varphi \subseteq (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$:

$$R_\varphi := \{(\mathbf{0}, \mathbf{0})\} \quad \cup \quad \{(\mathbf{0}, d) \mid d \in \varphi(\emptyset)\} \quad \cup \quad \{(d_1, d_2) \mid d_2 \in \varphi(\{d_1\}) \setminus \varphi(\emptyset)\}.$$

The intuition is that, in order to specify the union-distributive function φ, it suffices to fix $\varphi(\emptyset)$ and $\varphi(\{d\})$ for every $d \in D$. Then, we always have

$$\varphi(\{d_1, d_2, \ldots, d_r\}) = \varphi(\{d_1\}) \cup \varphi(\{d_2\}) \cup \cdots \cup \varphi(\{d_r\}).$$

We use a new item $\mathbf{0}$ to model $\varphi(\emptyset)$, i.e. $\mathbf{0}\ R_\varphi\ d \Leftrightarrow d \in \varphi(\emptyset)$. To specify $\varphi(\{d\})$, we first note that $\varphi(\emptyset) \subseteq \varphi(\{d\})$, so we only need to specify the elements that are in $\varphi(\{d\})$ but not $\varphi(\emptyset)$. These are precisely the elements that are in relation with d. In other words, $\varphi(\{d\}) = \varphi(\emptyset) \cup \{d' \mid d\ R_\varphi\ d'\}$. We can further represent the relation R_φ as a bipartite graph H_φ in which each part consists of the vertices $D \cup \{\mathbf{0}\}$ and R_φ defines the edges.

Example. Figure 2 shows the graph representation of several union-distributive functions.

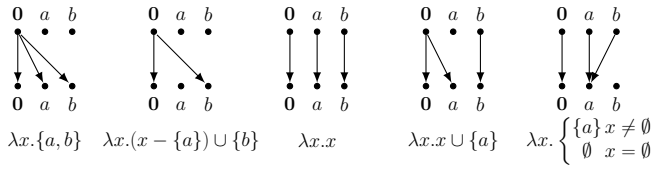

Fig. 2. Graph representation of union-distributive functions with $D = \{a, b\}$ [17].

Composition of Graph Representations [53]. What makes this graph representation particularly elegant is that we can compose two functions by a simple reachability computation. Specifically, if φ_1 and φ_2 are distributive, then so is $\varphi_2 \circ \varphi_1$. By definition chasing, we can see that $R_{\varphi_2 \circ \varphi_1} = R_{\varphi_1}; R_{\varphi_2} = \{(d_1, d_2) \mid \exists d_3\ \ (d_1, d_3) \in R_{\varphi_1} \wedge (d_3, d_2) \in R_{\varphi_2}\}$. Thus, to compute the graph representation $H_{\varphi_2 \circ \varphi_1}$, we simply merge the bottom part of H_{φ_1} with the top part of H_{φ_2} and then compute reachability from the top-most layer to the bottom-most layer.

Example. Figure 3 illustrates how the composition of two distributive functions can be obtained using their graph representations. Note that this process sometimes leads to superfluous edges. For example, since we have the edge $(\mathbf{0}, a)$ in the result, the edge (b, a) is not necessary. However, having it has no negative side-effects, either.

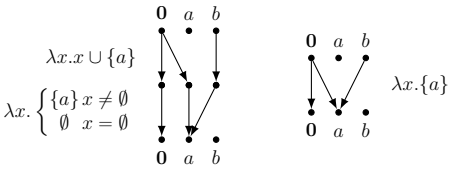

Fig. 3. Composing two distributive functions using reachability [17].

Exploded Supergraph [53]. Consider an IFDS arena $(G = (V, E), D, \Phi, M, \cup)$ as above and let $D^* := D \cup \{\mathbf{0}\}$. The *exploded supergraph* of this arena is a directed graph $\overline{G} = (\overline{V}, \overline{E})$ in which:

- $\overline{V} = V \times D^*$, i.e. we take $|D^*|$ copies of each vertex in the supergraph G, one corresponding to each data fact in D^*.

- $\overline{E} = \{(u_1, d_1, u_2, d_2) \in \overline{V} \times \overline{V} \mid (u_1, u_2) \in E \ \wedge \ (d_1, d_2) \in R_{M(u_1,u_2)}\}$. In other words, every edge between vertices u_1 and u_2 in the supergraph G is now replaced by the graphic representation of its corresponding distributive flow function $M(u_1, u_2)$.

Naturally, we say a path $\overline{\Pi}$ in $\overline{G}$ is interprocedurally (same-context) valid, if the path Π in G, obtained by ignoring the second component of every vertex in $\overline{\Pi}$, is interprocedurally (same-context) valid.

Reduction to Reachability. We can now reformulate our problem based on reachability by valid paths in the exploded supergraph $\overline{G}$. Consider an initial state $(u, D_1) \in V \times 2^D$ of the program and let $v \in V$ be another line. Since the exploded supergraph contains representations of all distributive flow functions, it already encodes the changes that happen to the data facts when we execute one step of the program. Thus, it is straightforward to see that for any data fact d_2, we have $d_2 \in \texttt{MIVP}(u, D_1, v)$ if and only if there exist a data fact $d_1 \in D_1$ such that the vertex (v, d_2) in $\overline{G}$ is reachable from the vertex (u, d_1) using an interprocedurally valid path [53]. Hence, our data-flow analysis is now reduced to reachability by valid paths. Moreover, instead of computing `MIVP` values, we can simplify our query structure so that each query provides two vertices (u, d_1) and (v, d_2) in the exploded supergraph $\overline{G}$ and asks whether there is a valid path from (u, d_1) to (v, d_2).

Example. Figure 4 shows the same program as in Figure 1, together with its exploded supergraph for null-pointer analysis. Here, we have two data facts: d_1 models the fact "the pointer `a` may be null" and d_2 does the same for `b`. Starting from line 5, i.e. the beginning of the `main` function, and knowing no data facts, i.e. $D_1 = \{\mathbf{0}\}$, we would like to see if either `a` or `b` might be null at the end of the `main` function. Using a reachability analysis on the exploded supergraph, we can identify all vertices that can be reached by a valid path (green) and conclude that neither `a` nor `b` may be null by the end of the program.

On-demand Analysis. As mentioned in Section 1, we focus on on-demand analysis and distinguish between a preprocessing phase in which the algorithm can perform a lightweight pass over the input and a query phase in which the algorithm has to respond to a large number of queries. The queries appear in a stream and the algorithm has to handle each query as fast as possible. Based on the discussion above, each query is of the form $(u_1, d_1, u_2, d_2) \in V \times D^* \times V \times D^*$ and the algorithm should report whether there exist an interprocedurally valid path from (u_1, d_1) to (u_2, d_2) in the exploded supergraph $\overline{G}$.

Bounded Bandwidth Assumption. Following previous works such as [53,17,31], we assume that the "bandwidth" in function calls and returns is bounded. More concretely, we assume there exists a small constant β such that for every interprocedural call-to-start or exit-to-return-site edge e in our supergraph G, the degree of each vertex in the graph representation $H_{M(e)}$ is at most β. This is a classical assumption made in IFDS and all of its extensions. Intuitively, it models the idea that every parameter in a called function depends on only a few variables in the call site line c of the callee, and conversely, that the return value of a function is only dependent on a few variables at its last line.

```
1  void g(int *&a, int *&b) {
2      b = a;
3  }
4
5  int main() {
6      int *a, *b;
7      a = new int(42);
8      g(a, b);
9      *b = 0;
10 }
```

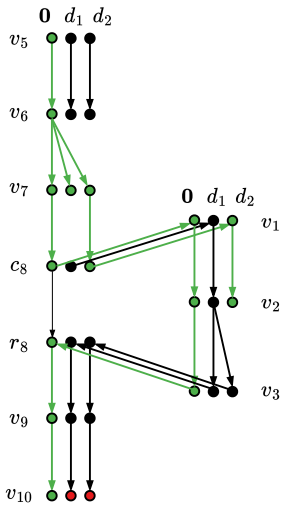

Fig. 4. A program (left) and its exploded supergraph (right).

3 Treewidth and Treedepth

In this section, we provide a short overview of the concepts of treewidth and treedepth. Treewidth and treedepth are both graph sparsity parameters and we will use them in our algorithm in the next section to formalize the sparsity of control-flow graphs and call graphs, respectively.

Tree Decompositions [55,54,10]. Given an undirected graph $G = (V, E)$, a *tree decomposition* of G is a rooted tree $T = (\mathfrak{B}, E_T)$ such that:

i. Every node $b \in \mathfrak{B}$ of the tree T has a corresponding subset $V_b \subseteq V$ of vertices of G. To avoid confusion, we reserve the word *"vertex"* for vertices of G and use the word *"bag"* to refer to nodes of the tree T. This is natural, since each bag b has a subset V_b of vertices.
ii. Every vertex appears in some bag, i.e. $\bigcup_{b \in \mathfrak{B}} V_b = V$.
iii. For every edge $\{u, v\} \in E$, there is a bag that contains both of its endpoints, i.e. $\exists b \in \mathfrak{B} \quad \{u, v\} \subseteq V_b$.
iv. Every vertex $v \in V$ appears in a connected subtree of T. Equivalently, if b is on the unique path from b' to b'' in T, then $V_b \supseteq V_{b'} \cap V_{b''}$.

When talking about tree decompositions of directed graphs, we simply ignore the orientation of the edges and consider decompositions of the underlying undirected graph. Intuitively, a tree decomposition covers the graph G by a number of bags[2] that are connected to each other in a tree-like manner. If the bags are small, we are then able to perform dynamic programming on G in a very similar manner to trees [8]. This is the motivation behind the following definition.

[2] The bags do not have to be disjoint.

Treewidth [55]. The *width* of a tree decomposition is defined as the size of its largest bag minus 1, i.e. $w(T) := \max_{b \in \mathfrak{B}} |V_b| - 1$. The *treewidth* of a graph G is the smallest width amongst all of its tree decompositions. Informally speaking, treewidth is a measure of tree-likeness. Only trees and forests have a treewidth of 1, and, if a graph G has treewidth k, then it can be decomposed into bags of size at most $k + 1$ that are connected to each other in a tree-like manner.

Example. Figure 5 shows a graph G on the left and a tree decomposition of width 2 for G on the right. In the tree decomposition, we have highlighted the connected subtree of each vertex by dotted lines. This tree decomposition is optimal and hence the treewidth of G is 2.

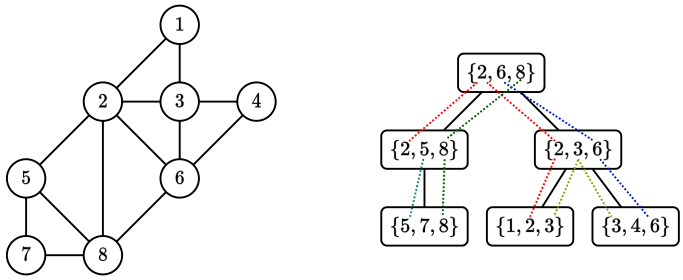

Fig. 5. A graph G (left) and one of its tree decompositions (right).

Computing Treewidth. In general, it is NP-hard to compute the treewidth of a given graph. However, for any constant k, there is a linear-time algorithm that decides whether the graph has treewidth at most k and, if so, also computes an optimal tree decomposition [9]. As such, most treewidth-based algorithms assume that an optimal tree decomposition is given as part of the input.

Treewidth of Control-flow Graphs. In [62], it was shown that the control-flow graphs of `goto`-free programs in a number of languages such as C and Pascal have a treewidth of at most 7. Moreover, [62] also provides a linear-time algorithm that, while not necessarily optimal, always outputs a tree decomposition of width at most 7 for the control-flow graph of programs in these languages by a single pass over the parse tree of the program. This is the algorithm we use for obtaining our tree decompositions in Section 5. Alternatively, one can use the algorithm of [9] to ensure that an optimal decomposition is used at all times. The theoretical bound of [62] does not apply to Java, but the work [30] showed that the treewidth of control-flow graphs in real-world Java programs is also bounded. Nevertheless, one can theoretically construct pathological examples with high treewidth.

Balancing Tree Decompositions. Finally, the runtime of our algorithm in Section 4 depends on the height of the tree decomposition. However, [12] provides a linear-time algorithm that, given a graph G and a tree decomposition

of constant width t, produces a binary tree decomposition of height $O(\lg n)$ and width $O(t)$. Combining this with the algorithms of [62] and [9] for computing low-width tree decompositions allows us to assume that we are always given a balanced and binary tree decomposition of bounded width for each one of our control-flow graphs as part of our IFDS input.

We now switch our focus to the second parameter that appears in our algorithms, namely treedepth.

Partial Order Trees [45]. Let $G = (V, E)$ be an undirected connected graph. A *partial order tree* (POT)[3] over G is a rooted tree $T = (V, E_T)$ on the same set of vertices as G that additionally satisfies the following property:

- For every edge $\{u, v\} \in E$ of G, either u is an ancestor of v in T or v is ancestor of u in T.

The intuition is quite straightforward: T defines a partial order $\prec_T$ over the vertices V in which every element u is assumed to be smaller than its parent p_u, i.e. $u \prec_T p_u$. For T to be a valid POT, every pair of vertices that are connected by an edge in G should be comparable in $\prec_T$. If G is not connected, then we will have a partial order forest, consisting of a partial order tree for each connected component of G. With a slight abuse of notation, we call this a POT, too.

Example. Figure 6 shows a graph G (left) together with a POT of depth 4 for G (right). In the POT, the edges of the original graph G are shown by dotted red lines. Every edge of G goes from a node in T to one of its ancestors.

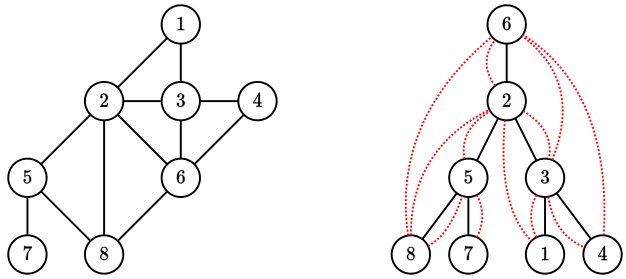

Fig. 6. A graph G (left) and a POT of depth 4 for G (right).

Treedepth [45]. The treedepth of an undirected graph G is the smallest depth among all POTs of G.

Path Property of POTs [45]. Let $T = (V, E_T)$ be a POT for a graph $G = (V, E)$ and u and v two vertices in V. Define A_u as the set of ancestors of u in T and define A_v similarly. Let $A := A_u \cap A_v$ be the set of common ancestors of u and v. Then, any path that goes from u to v in the graph G has to intersect A.

[3] The name *partial order tree* is not standard in this context, but we use it throughout this work since it provides a good intuition about the nature of T.

Sparsity Assumption. In the sequel, our algorithm is going to assume that call graphs of real-world programs have small treedepth. We establish this experimentally in Section 5. However, there is also a natural reason why this assumption is likely to hold in practice. Consider the functions in a program. It is natural to assume that they were developed in a chronological order, starting with base (phase 1) functions, and then each phase of the project used the functions developed in the previous phases as libraries. Thus, the call graph can be partitioned to a small number of layers based on the development phase of each function. Moreover, each function typically calls only a small number of previous functions. So, an ordering based on development phase is likely to give us a POT with small depth. The depth would typically depend on the number of phases and the degree of each function in the call graph, but these are both small parameters in practice.

Computing Treedepth. As in the case of treewidth, it is NP-hard to compute the treedepth of a given graph [48]. However, for any fixed constant k, there is a linear-time algorithm that decides whether a given graph has treedepth at most k and, if so, produces an optimal POT [41]. Thus, in the sequel, we assume that all inputs include a POT of the call graph with bounded depth.

4 Our Parameterized Algorithm

In this section, we present our parameterized algorithm for solving the general case of IFDS data-flow analysis, assuming that the control-flow graphs have bounded treewidth and the call graph has bounded treedepth. Throughout this section, we fix an IFDS arena $(G, D, \Phi, M, \cup)$ given by an exploded supergraph $\overline{G}$. Finally, we assume that every control-flow graph comes with a balanced binary tree decomposition of width at most k_1. We also assume that a POT of depth k_2 over the call graph is given as part of the input. All these assumptions are without loss of generality since the tree decompositions and POT can be computed in linear time using the algorithms mentioned in Section 3. Before presenting our algorithm, we should first define a few useful notions.

Algorithm for Same-Context IFDS. The work [17] provides an on-demand parameterized algorithm for same-context IFDS. This algorithm requires a balanced and binary tree decomposition of constant width for every control-flow graph and provides a preprocessing runtime of $O(n \cdot |D|^3)$, after which it can answer *same-context* queries in time $O\left(\lceil \frac{|D|}{\lg n} \rceil\right)$. A same-context query is a query of the form $(u_1, d_1, u_2, d_2) \in V \times D^* \times V \times D^*$ which asks whether there exists a *same-context* valid path from (u_1, d_1) to (u_2, d_2) in the exploded supergraph $\overline{G}$. In our algorithm, we use [17]'s algorithm for same-context queries as a black box.

Stack States. Let F be the set of functions in our program. A *stack state* is simply a finite sequence of functions $\xi = \langle \xi_i \rangle_{i=1}^{s} \in F^s$. We use a stack state to keep track of the set of functions that have been called but have not finished their execution and returned.

Persistence. Consider an interprocedurally valid path $\Pi = \langle \pi_i \rangle_{i=1}^{p}$ in the supergraph G and let $\Pi^* = \langle \pi_i^* \rangle_{i=1}^{s}$ be the sub-sequence of Π that only includes call vertices c_l and return vertices r_l. For each π_i^* that is a call vertex, let f_i be the function called by π_i^*. We say the function call to f_i is *temporary* if π_i^* is matched by a corresponding return-site vertex π_j^* in Π^* with $j > i$. Otherwise, f_i is is a *persistent* function call. In other words, temporary function calls are the ones who return before the end of the path Π and persistent ones are those that are added to the stack but never popped. So, if the stack is at state ξ before executing Π, it will be in state $\xi \cdot \langle f_{i_1} \cdot f_{i_2} \cdots f_{i_r} \rangle$ after Π's execution, in which the f_{i_j}'s are our persistent function calls. Moreover, we can break down the path Π as follows:

$$\Pi = \Sigma_0 \cdot \Sigma_1 \cdot \pi_{i_1} \cdot \Sigma_2 \cdot \pi_{i_2} \cdots \Sigma_r \cdot \pi_{i_r} \cdot \Sigma_{r+1} \tag{1}$$

in which Σ_0 is an *intraprocedural* path, i.e. the part of Π that does not leave the initial function. Note that we either have $\Pi = \Sigma_0$ or Σ_0 should end with a function call. For every $i \neq 0$, Σ_i is a same-context valid path from the starting point of a function and π_{i_j} is a call vertex that calls the next persistent function f_{i_j}. We call (1) the *canonical partition* of the path Π.

Exploded Call Graph. Let $C = (F, E_C)$ be the call graph of our IFDS instance, in which F is the set of functions in the program. We define the *exploded call graph* $\overline{C} = (\overline{F}, \overline{E_C})$ as follows:

- Our vertex set $\overline{F}$ is simply $F \times D^*$. Recall that $D^* := D \cup \{\mathbf{0}\}$.
- There is an edge from the vertex (f_1, d_1) to the vertex (f_2, d_2) in $\overline{E_C}$ iff:
  - There is a call statement $c \in V$ in the function f_1 that calls f_2;
  - There exist a data fact $d_3 \in D^*$ such that (i) there is a *same-context* valid path from (s_{f_1}, d_1) to (c, d_3) in the exploded supergraph $\overline{G}$, and (ii) there is an edge from (c, d_3) to (s_{f_2}, d_2) in the exploded supergraph $\overline{G}$.

The edges of the exploded call graph model the effect of a valid path that starts at s_{f_1}, i.e. the first line of f_1, when the function call stack is empty and reaches s_{f_2}, with stack state $\langle f_2 \rangle$. Informally, this corresponds to executing the program starting form f_1, potentially calling any number of temporary functions, then waiting for all of these temporary functions and their children to return so that we again have an empty stack, and then finally calling f_2 from the call-site c, hence reaching stack state $\langle f_2 \rangle$. Intuitively, this whole process models the substring $\Sigma \cdot c$ in the canonical partition of a valid path, in which Σ is a same-context valid path, and f_2 is the next persistent function, which was called at c. Hence, going forward, we do not plan to pop f_2 from the stack.

Treedepth of $\overline{C}$. Recall that we have a POT T of depth k_2 for the call graph C. In $\overline{C}$, every $f \in C$ is replaced by $|D^*|$ vertices $(f, \mathbf{0}), (f, d_1), \ldots, (f, d_{|D|})$. We can obtain a valid POT $\overline{T}$ for $\overline{C}$ by processing the tree in a top-down order and replacing every vertex that corresponds to a function f with a path of length $|D^*|$, as shown in Figure 7. It is straightforward to verify that $\overline{T}$ is a valid POT of depth $k_2 \cdot |D^*|$ for $\overline{C}$.

Preprocessing. The preprocessing phase of our algorithm consists of the following four steps:

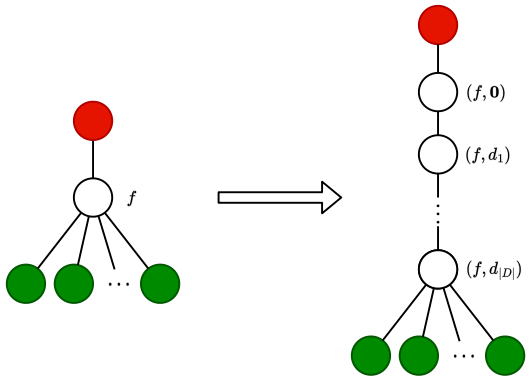

Fig. 7. Obtaining $\overline{T}$ from T by expanding each vertex to a path.

1. *Same-context Preprocessing:* Our algorithm runs the preprocessing phase of [17]'s algorithm for same-context IFDS. This is done as a black box. See [17] for the details of this step.
2. *Intraprocedural Preprocessing:* For every vertex $(u, d) \in \overline{G}$, for which u is a line of the program in the function f, our algorithm performs an *intraprocedural* reachability analysis and finds a list of all the vertices of the form (c, d') such that:
   - c is a call-site vertex in the same function f.
   - There is an *intraprocedural* path from (u, d) to (c, d') that always remains within f and does not cause any function calls.

   Our algorithm computes this by a simple reverse DFS from every (c, d').
3. *Computing Exploded Call Graph:* Our algorithm generates the exploded call graph $\overline{C}$ using its definition above. It iterates over every function f_1 and call site c in f_1. Let f_2 be the function called at c. For every pair $(d_1, d_3) \in D^* \times D^*$, our algorithm queries the same-context IFDS algorithm of [17] to see if there is a same-context valid path from (s_{f_1}, d_1) to (c, d_3). Note that we can make such queries since we have already performed the required same-context preprocessing in Step 1 above. If the query's result is positive, the algorithm iterates over every $d_2 \in D^*$ such that (c, d_3, s_{f_2}, d_2) is an edge in the exploded supergraph $\overline{G}$, and adds an edge from (f_1, d_1) to (f_2, d_2) in $\overline{C}$. The algorithm also computes the POT $\overline{T}$ as mentioned above.
4. *Computing Ancestral Reachability in $\overline{T}$:* For every vertex u in $\overline{T}$, let $\overline{T}_u^{\downarrow}$ be the subtree of $\overline{T}$ rooted at u and $\overline{F}_u^{\downarrow}$ be the set of descendants of u. For every u and every $v \in \overline{F}_u^{\downarrow}$, our algorithm precomputes $reach(u, v)$, i.e. whether u is reachable from v in $\overline{C}$ and also $reach(v, u)$.
   To compute this, for every vertex u and every descendant v of u, we define:

$$up[u, v] := \begin{cases} 1 & \text{there is a path from } v \text{ to } u \text{ in } \overline{C}[\overline{F}_u^{\downarrow}] \\ 0 & \text{otherwise} \end{cases},$$

$$down[u,v] := \begin{cases} 1 & \text{there is a path from } u \text{ to } v \text{ in } \overline{C}[\overline{F}_u^{\downarrow}] \\ 0 & \text{otherwise} \end{cases}.$$

Note that in the definition above, we are only considering paths whose every internal vertex is in the subtree of u. We can find the values of $down[u,v]$ by simply running a DFS from u but ignoring all the edges that leave the subtree $\overline{T}_u^{\downarrow}$. Similarly, we can find the values of $up[u,v]$ by a similar DFS in which the orientation of all edges are reversed.
By the path property of POTs, every path ρ from v to u in $\overline{C}$ either has all of its vertices in the subtree $\overline{T}_u^{\downarrow}$ or visits some ancestors of u as internal vertices. Let w be the highest ancestor of u that is visited by ρ. Then, we must have $up[w,v] = down[w,u] = 1$. Similarly, if there is a path from u to v, we must have $up[w,u] = down[w,v] = 1$. Our algorithm simply sets:

$$reach(u,v) = \bigvee_w \left(up[w,u] \wedge down[w,v]\right),$$

and

$$reach(v,u) = \bigvee_w \left(up[w,v] \wedge down[w,u]\right).$$

Query. After the end of the preprocessing phase, our algorithm is ready to accept queries. Suppose that a query q asks whether there exists a valid interprocedural path from (u_1,d_1) to (u_2,d_2) in $\overline{G}$. Suppose that $\overline{\Pi}$ is such a valid path and Π is its trace on the supergraph G, i.e. the path obtained from $\overline{\Pi}$ by ignoring the second component of every vertex. We consider the canonical partition of Π as

$$\Pi = \Sigma_0 \cdot (\Sigma_1 \cdot \pi_{i_1}) \cdot (\Sigma_2 \cdot \pi_{i_2}) \cdots (\Sigma_r \cdot \pi_{i_r}) \cdot \Sigma_{r+1}$$

and its counterpart in $\overline{\Pi}$ as

$$\overline{\Pi} = \overline{\Sigma_0} \cdot \left(\overline{\Sigma_1} \cdot \overline{\pi_{i_1}}\right) \cdot \left(\overline{\Sigma_2} \cdot \overline{\pi_{i_2}}\right) \cdots \left(\overline{\Sigma_r} \cdot \overline{\pi_{i_r}}\right) \cdot \overline{\Sigma_{r+1}}.$$

Let $\overline{\Sigma_j}[1]$ be the first vertex in $\overline{\Sigma_j}$. For every $j \geq 1$, consider the subpath

$$\overline{\Sigma_j} \cdot \overline{\pi_{i_j}} \cdot \overline{\Sigma_{j+1}}[1].$$

This subpath starts at the starting point s_f of some function f and ends at the starting point $s_{f'}$ of the function f' called in $\overline{\pi_{i_j}}$. Thus, it goes from a vertex of the form (s_f, d_1) to a vertex of the form $(s_{f'}, d_2)$. However, by the definition of our exploded call graph $\overline{C}$, we must have an edge $\overline{e_j}$ in $\overline{C}$ going from (f,d_1) to (f',d_2). With a minor abuse of notation, we do not differentiate between f and s_f and replace this subpath with $\overline{e_j}$. Hence, every interprocedurally valid $\overline{\Pi}$ can be partitioned in the following format:

$$\overline{\Pi} = \overline{\Sigma_0} \cdot \overline{e_1} \cdot \overline{e_2} \cdots \overline{e_r} \cdot \overline{\Sigma_{r+1}}.$$

In other words, to obtain an interprocedurally valid path, we should first take an intraprocedural path $\overline{\Sigma_0}$ in our initial function, followed by a path $\overline{e_1} \cdot \overline{e_2} \cdots \overline{e_r}$ in

the exploded call graph $\overline{C}$, and then a same-context valid path $\overline{\Sigma_{r+1}}$ in our target function. Note that $\overline{\Sigma_{r+1}}$ begins at the starting point of our target function.

Our algorithm uses the observation above to answer the queries. Recall that the query q is asking whether there exists a path from (u_1, d_1) to (u_2, d_2) in $\overline{G}$. Let f_1 be the function of u_1 and f_2 the function containing u_2. Our algorithm performs the following steps to answer the query:

1. Take all vertices of the form (c, d_3) such that c is a call vertex in f_1 and (c, d_3) is intraprocedurally reachable from (u_1, d_1). Note that this was precomputed in Step 2 of our preprocessing.
2. Find all successors of the vertices in Step 1 in $\overline{G}$. These successors are all of the form $(s_{f'}, d_4)$ for some function f', and their corresponding nodes in the exploded call graph are of the form (f', d_4).
3. Compute the set of all (f_2, d_5) vertices in $\overline{C}$ that are reachable from one of the (f', d_4) vertices obtained in the previous step. In this case, the algorithm uses the path property of POTs and tries all possible common ancestors of (f_2, d_5) and (f', d_4) as potential internal vertices in the path.
4. For each (f_2, d_5) found in the previous step, ask the same-context query from (s_{f_2}, d_5) to (f_2, d_2). For these same-context queries, our algorithm uses the method of [17] as a black box.
5. If any of the same-context queries in the previous step return true, then our algorithm also answers true to the query q. Otherwise, it answers false.

Figure 8 provides an overview of how our query phase breaks an interprocedurally valid path down between $\overline{G}$ (red) and $\overline{C}$ (blue). Note that we do not distinguish between the vertex (f_2, d_5) of $\overline{C}$ and vertex (s_{f_2}, d_5) of $\overline{G}$.

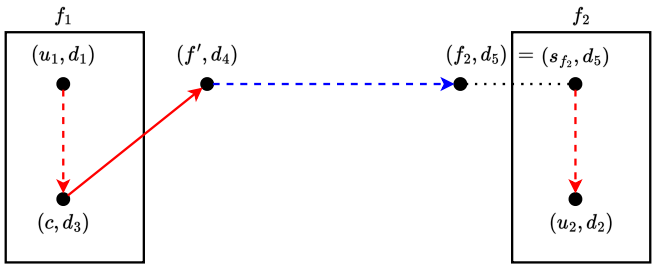

Fig. 8. An overview of the query phase.

Runtime Analysis. Our algorithm is much faster than the classical IFDS algorithm of [53]. More specifically, for the preprocessing, we have:

- Step 1 is a black box from [17] and takes $O(n \cdot |D|^3)$.
- Step 2 is a simple intraprocedural analysis that runs a reverse DFS from every node (c, d) in any function f. Assuming that the function f has α lines of code and a total of β function call statements, this will take $O(\alpha \cdot \beta \cdot |D|^3)$. Assuming that β is a small constant, this leads to an overall runtime of $O(n \cdot |D|^3)$.
- In Step 3, we have at most $O(n \cdot |D|)$ call nodes of the form (c, d_3). Based on the bounded bandwidth assumption, each such node leads to constantly

many possibilities for d_2. So, we perform at most $O(n \cdot |D|^2)$ calls to the same-context query procedure. Each same-context query takes $O(\lceil |D| / \lg n \rceil)$, so the overall runtime of this step is $O(n \cdot |D|^3 / \lg n)$.
- In Step 4, the total time for computing all the *up* and *down* values is $O(n \cdot |D|^3 \cdot k_2)$. This is because $\overline{C}$ has at most $O(n \cdot |D|)$ vertices and $O(n \cdot |D|^2)$ edges and each edge can be traversed at most $O(|D| \cdot k_2)$ times in the DFS, where k_2 is the depth of our POT for C. Note that the treedepth of $\overline{C}$ is a factor $|D|$ larger than that of C. Finally, computing the *reach* values takes $O(n \cdot |D|^3 \cdot k_2^2)$ time.

Therefore, the total runtime of our preprocessing phase is $O(n \cdot |D|^3 \cdot k_2^2)$, which has only linear dependence on the number of lines, n.

To analyze the runtime of a query, note that there are $O(\beta \times |D|)$ different possibilities for (c, d_3). Due to the bounded bandwidth assumption, each of these correspond to a constant number of (f', d_4)'s. For each (f', d_4) and (f_2, d_5), we should perform a reachability query using the POT $\overline{T}$. So, we might have to try up to $O(k_2 \cdot |D|)$ common ancestors. So, the total runtime for finding all the (f_2, d_5)'s is $O(|D|^3 \cdot k_2 \cdot \beta)$. Finally, we have to perform a same-context query from every (s_{f_2}, d_5) to (u_2, d_2). So, we do a total of at most $O(|D|)$ queries, each of which take $O(|D|)$. So, the total runtime is $O(|D|^3 \cdot k_2 \cdot \beta)$, which is equal to $O(|D|^3)$ in most real-world scenarios where k_2 and β are small constants.

5 Experimental Results

Implementation and Machine. We implemented our algorithm in a combination of C++ and Java, and used the Soot [63] framework to obtain the control-flow and call graphs. To compute treewidth and treedepth, we used the winning open-source tools submitted to past PACE challenges [23,37]. All experiments were run on an Intel i7-11800H machine (2.30 GHz, 8 cores, 16 threads) with 12 GB of RAM.

Benchmarks and Experimental Setup. We compare the performance of our method against the standard IFDS algorithm [53] and its on-demand variant [31] and use the standard DaCapo benchmarks [5] as input programs. These are real-world programs with hundreds of thousands of lines of code. For each benchmark, we consider three different classical data-flow analyses: (i) reachability analysis for dead-code elimination, (ii) null-pointer analysis, and (iii) possibly-uninitialized variables analysis. For each analysis, we gave each of the algorithms 10 minutes time over each benchmark and recorded the number of queries that the algorithm successfully handled in this time. The queries themselves were randomly generated and the number of queries was also limited to n, i.e. the number of lines in the code. We then report the average cost of each query, i.e. the algorithms total runtime divided by the number of queries it could handle. The reason for this particular setup is that [53] and [31] do not distinguish between preprocessing and query. So, to avoid giving our own method any undue advantage, we have to include both our preprocessing and our query time in the mix.

Treewidth and Treedepth. In our experiments, the maximum encountered treewidth was 10, whereas the average was 9.1. Moreover, the maximum treedepth

was 135 and the average was 43.8. Hence, our central hypothesis that real-world programs have small treewidth and treedepth holds in practice and the widths and depths are much smaller than the number of lines in the program.

Results. Figure 9 provides the average query time for each analysis. Each dot corresponds to one benchmark. We use `PARAM`, `IFDS` and `DEM` to refer to our algorithm, the IFDS algorithm in [53], and the on-demand IFDS algorithm in [31], respectively. The reported instance sizes are the number of edges $\overline{G}$.

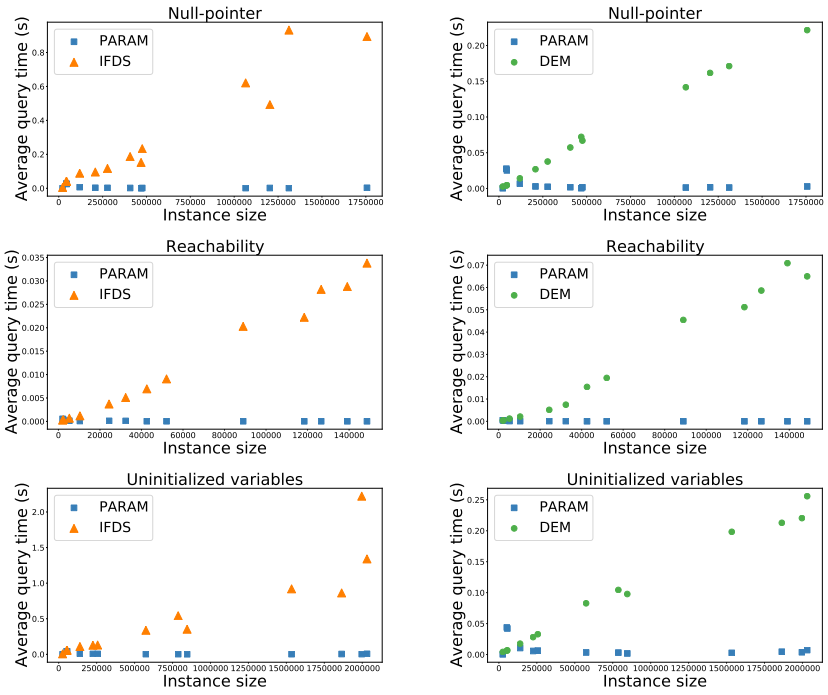

Fig. 9. Comparison of the average cost per query for our algorithm vs [53] and [31].

Discussion. As shown in Section 4, our algorithm's preprocessing has only linear dependence on the number n of lines and our query time is completely independent of n. Thus, our algorithm has successfully pushed most of the time complexity on the small parameters such as the treewidth k_1, treedepth k_2, bandwidth b and maximum number of function calls in each function, i.e. β. All these parameters are small constants in practice. Specifically, the two most important ones are always small. The treewidth in DaCapo benchmarks never exceeds 10 and the treedepth is at most 135. This is in contrast to n which is the hundreds of thousands and the instance size, which can be up to around $2 \cdot 10^6$. In contrast, both [53] and [31] have a quadratic dependence on n. Unsurprisingly, this leads to a huge gap in the practical runtimes and our algorithm is on average faster than the best among [53] and [31] by a factor of 158, i.e. more than two orders of magnitude. Moreover, the difference is much starker on larger benchmarks, in which the ratio of our parameters to n is close to 0.

References

1. T.J. Watson libraries for analysis, with frontends for Java, Android, and JavaScript, and many common static program analyses, `https://github.com/wala/WALA`
2. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.D.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: PLDI. pp. 259–269 (2014)
3. Babich, W.A., Jazayeri, M.: The method of attributes for data flow analysis: Part II. demand analysis. Acta Informatica **10**, 265–272 (1978)
4. Bebenita, M., Brandner, F., Fähndrich, M., Logozzo, F., Schulte, W., Tillmann, N., Venter, H.: SPUR: a trace-based JIT compiler for CIL. In: OOPSLA. pp. 708–725 (2010)
5. Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H.B., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA. pp. 169–190 (2006)
6. Bodden, E.: Inter-procedural data-flow analysis with IFDS/IDE and soot. In: SOAP. pp. 3–8 (2012)
7. Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M.: SPLLIFT: statically analyzing software product lines in minutes instead of years. In: PLDI. pp. 355–364 (2013)
8. Bodlaender, H.L.: Dynamic programming on graphs with bounded treewidth. In: ICALP. vol. 317, pp. 105–118 (1988)
9. Bodlaender, H.L.: A linear time algorithm for finding tree-decompositions of small treewidth. In: STOC. pp. 226–234 (1993)
10. Bodlaender, H.L.: A tourist guide through treewidth. Acta Cybern. **11**(1-2), 1–21 (1993)
11. Bodlaender, H.L., Deogun, J.S., Jansen, K., Kloks, T., Kratsch, D., Müller, H., Tuza, Z.: Rankings of graphs. SIAM J. Discret. Math. **11**(1), 168–181 (1998)
12. Bodlaender, H.L., Hagerup, T.: Parallel algorithms with optimal speedup for bounded treewidth. SIAM Journal on Computing **27**(6), 1725–1746 (1998)
13. Burgstaller, B., Blieberger, J., Scholz, B.: On the tree width of Ada programs. In: Ada-Europe. vol. 3063, pp. 78–90 (2004)
14. Callahan, D., Cooper, K.D., Kennedy, K., Torczon, L.: Interprocedural constant propagation. In: CC. pp. 152–161 (1986)
15. Chang, W., Streiff, B., Lin, C.: Efficient and extensible security enforcement using dynamic data flow analysis. In: CCS. pp. 39–50 (2008)
16. Chatterjee, K., Goharshady, A.K., Goharshady, E.K.: The treewidth of smart contracts. In: SAC. pp. 400–408 (2019)
17. Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Pavlogiannis, A.: Optimal and perfectly parallel algorithms for on-demand data-flow analysis. In: ESOP. vol. 12075, pp. 112–140 (2020)
18. Chen, T., Lin, J., Dai, X., Hsu, W., Yew, P.: Data dependence profiling for speculative optimizations. In: CC. vol. 2985, pp. 57–72 (2004)
19. Chow, A.L., Rudmik, A.: The design of a data flow analyzer. In: CC. pp. 106–113 (1982)
20. Collard, J.F., Knoop, J.: A comparative study of reaching-definitions analyses (1998)
21. Dangel, A., Fournier, C., et al.: PMD Eclipse plugin, `https://github.com/pmd/pmd-eclipse-plugin`
22. Das, A., Lal, A.: Precise null pointer analysis through global value numbering. In: ATVA. vol. 10482, pp. 25–41 (2017)

23. Dell, H., Komusiewicz, C., Talmon, N., Weller, M.: The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In: IPEC. pp. 30:1–30:12 (2018)
24. Duesterwald, E., Gupta, R., Soffa, M.L.: Demand-driven computation of interprocedural data flow. In: POPL. pp. 37–48 (1995)
25. Eclipse Foundation: Eclipse documentation, Java development user guide, `http://help.eclipse.org/2022-06/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/preferences/java/compiler/ref-preferences-errors-warnings.htm`
26. Flückiger, O., Scherer, G., Yee, M., Goel, A., Ahmed, A., Vitek, J.: Correctness of speculative optimizations with dynamic deoptimization. In: POPL. pp. 49:1–49:28 (2018)
27. Gould, C., Su, Z., Devanbu, P.T.: JDBC checker: A static analysis tool for SQL/JDBC applications. In: ICSE. pp. 697–698 (2004)
28. Grove, D., Torczon, L.: Interprocedural constant propagation: A study of jump function implementations. In: PLDI. pp. 90–99 (1993)
29. Gupta, R., Benson, D., Fang, J.Z.: Path profile guided partial dead code elimination using predication. In: PACT. pp. 102–113 (1997)
30. Gustedt, J., Mæhle, O.A., Telle, J.A.: The treewidth of Java programs. In: ALENEX. vol. 2409, pp. 86–97 (2002)
31. Horwitz, S., Reps, T.W., Sagiv, S.: Demand interprocedural dataflow analysis. In: FSE. pp. 104–115 (1995)
32. Khedker, U., Sanyal, A., Sathe, B.: Data flow analysis: theory and practice. CRC Press (2017)
33. Kildall, G.A.: A unified approach to global program optimization. In: POPL. pp. 194–206 (1973)
34. Kildall, G.A.: Global expression optimization during compilation. University of Washington (1972)
35. Knoop, J., Steffen, B.: Efficient and optimal bit-vector data flow analyses: A uniform interprocedural framework. Institut für Informatik und Praktische Mathematik Kiel: Bericht (1993)
36. Knoop, J., Steffen, B., Vollmer, J.: Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. TOPLAS **18**(3), 268–299 (1996)
37. Łukasz Kowalik, Mucha, M., Nadara, W., Pilipczuk, M., Sorge, M., Wygocki, P.: The PACE 2020 Parameterized Algorithms and Computational Experiments Challenge: Treedepth. In: IPEC. pp. 37:1–37:18 (2020)
38. Kurdahi, F.J., Parker, A.C.: REAL: a program for register allocation. In: DAC. pp. 210–215 (1987)
39. Lin, J., Chen, T., Hsu, W., Yew, P., Ju, R.D., Ngai, T., Chan, S.: A compiler framework for speculative optimizations. TACO **1**(3), 247–271 (2004)
40. Meyer, B.: Ending null pointer crashes. Commun. ACM **60**(5), 8–9 (2017)
41. Nadara, W., Pilipczuk, M., Smulewicz, M.: Computing treedepth in polynomial space and linear FPT time. CoRR **abs/2205.02656** (2022)
42. Naeem, N.A., Lhoták, O., Rodriguez, J.: Practical extensions to the IFDS algorithm. In: CC. vol. 6011, pp. 124–144 (2010)
43. Nanda, M.G., Sinha, S.: Accurate interprocedural null-dereference analysis for java. In: ICSE. pp. 133–143. IEEE (2009)
44. Nešetřil, J., De Mendez, P.O.: Sparsity: graphs, structures, and algorithms. Springer (2012)
45. Nesetril, J., de Mendez, P.O.: Tree-depth, subgraph coloring and homomorphism bounds. Eur. J. Comb. **27**(6), 1022–1041 (2006)
46. Nguyen, T.V.N., Irigoin, F., Ancourt, C., Coelho, F.: Automatic detection of uninitialized variables. In: CC. vol. 2622, pp. 217–231 (2003)

47. Pessoa, T., Monteiro, M.P., Bryton, S., et al.: An eclipse plugin to support code smells detection. arXiv preprint arXiv:1204.6492 (2012)
48. Pothen, A.: The complexity of optimal elimination trees. Tech. rep. (1988)
49. Rapoport, M., Lhoták, O., Tip, F.: Precise data flow analysis in the presence of correlated method calls. In: SAS. vol. 9291, pp. 54–71 (2015)
50. Reps, T.: Undecidability of context-sensitive data-dependence analysis. TOPLAS **22**(1), 162–186 (2000)
51. Reps, T.W.: Demand interprocedural program analysis using logic databases. In: ILPS. pp. 163–196 (1993)
52. Reps, T.W.: Program analysis via graph reachability. Inf. Softw. Technol. **40**(11-12), 701–726 (1998)
53. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL. pp. 49–61 (1995)
54. Robertson, N., Seymour, P.D.: Graph minors. iii. planar tree-width. Journal of Combinatorial Theory, Series B **36**(1), 49–64 (1984)
55. Robertson, N., Seymour, P.D.: Graph minors. ii. algorithmic aspects of tree-width. Journal of algorithms **7**(3), 309–322 (1986)
56. Rountev, A., Kagan, S., Marlowe, T.J.: Interprocedural dataflow analysis in the presence of large libraries. In: CC. vol. 3923, pp. 2–16 (2006)
57. Sagiv, S., Reps, T.W., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comput. Sci. **167**, 131–170 (1996)
58. Shang, L., Xie, X., Xue, J.: On-demand dynamic summary-based points-to analysis. In: CGO. pp. 264–274 (2012)
59. Späth, J., Ali, K., Bodden, E.: Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. In: POPL. pp. 48:1–48:29 (2019)
60. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: PLDI. pp. 387–400 (2006)
61. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: Demand-driven points-to analysis for Java. In: OOPSLA. pp. 59–76 (2005)
62. Thorup, M.: All structured programs have small tree-width and good register allocation. Inf. Comput. **142**(2), 159–181 (1998)
63. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: CASCON. p. 13. IBM (1999)
64. Xu, G., Rountev, A., Sridharan, M.: Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In: ECOOP. vol. 5653, pp. 98–122 (2009)
65. Yan, D., Xu, G., Rountev, A.: Demand-driven context-sensitive alias analysis for Java. In: ISSTA. pp. 155–165 (2011)
66. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: POPL. pp. 197–208 (2008)