Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2023-IC-004

2023-IC-004

DStream: A Streaming-Based Highly Parallel IFDS Framework

Xizao Wang, Zhiqiang Zuo, Lei Bu, Jianhua Zhao

Technical Report 2023

DStream: A Streaming-Based Highly Parallel IFDS Framework

Xizao Wang, Zhiqiang Zuo†, Lei Bu†, Jianhua Zhao
State Key Laboratory for Novel Software Technology, Nanjing University, China
wangxiz@smail.nju.edu.cn, zqzuo@nju.edu.cn, bulei@nju.edu.cn, zhaojh@nju.edu.cn

***Abstract*—The IFDS framework supports interprocedural dataflow analysis with distributive flow functions over finite domains. A large class of interprocedural dataflow analysis problems can be formulated as IFDS problems and thus can be solved with the IFDS framework precisely. Unfortunately, scaling IFDS analysis to large-scale programs is challenging in terms of both massive memory consumption and low analysis efficiency.**

This paper presents DStream, a scalable system dedicated to precise and highly parallel IFDS analysis for large-scale programs. DStream leverages a streaming-based out-of-core computation model to reduce memory footprint significantly and adopts fine-grained data parallelism to achieve efficiency. We implemented a taint analysis as a DStream instance analysis and compared DStream with three state-of-the-art tools. Our experiments validate that DStream outperforms all other tools with average speedups from 4.37x to 14.46x on a commodity PC with limited available memory. Meanwhile, the experiments confirm that DStream successfully scales to large-scale programs which the state-of-the-art tools (*e.g.*, FlowDroid and/or DiskDroid) fail to analyze.

***Index Terms*—interprocedural static analysis, IFDS analysis, streaming, data-parallel computation**

I. Introduction

The IFDS (interprocedural, finite, distributive, subset) framework, pioneered by Reps *et al.* [1], is a general interprocedural context-sensitive dataflow analysis framework. The framework is tailored for a rich set of problems that satisfy its restrictions (*a.k.a.*, the IFDS problems), including taint analysis [2], [3], [4], [5], program slicing [6], bug detection [7], [8], [9], [10], and security analysis [11]. The IFDS framework formulates an IFDS problem as a graph reachability problem that can be solved precisely and efficiently.

The IFDS framework is empowered by a tabulation algorithm (*i.e.*, the IFDS algorithm) of polynomial complexity. Although the IFDS algorithm is asymptotically fast, it suffers from poor scalability, especially when analyzing large-scale programs with expensive abstract domains. Such poor scalability mainly lies in two-fold: considerable time cost and massive memory consumption. Performance-wise, many studies over the past decades have been constantly proposed to accelerate the analysis by adopting sparse representations [3], incremental algorithms [12], and parallel accelerations [13], [14], [15]. However, the massive amount of peak memory consumption becomes a more severe bottleneck for scaling the IFDS analysis to large-scale programs. As reported by [16], analyzing even a moderate-sized program takes 730 GB memory for FlowDroid. SparseDroid [3] adopting sparse representation also consumes around 80 GB of RAM. Even the incremental IFDS solver Reviser [12] generally needs more than 35 GB memory for every run. How to tackle the memory obstacle is increasingly becoming crucial to scaling IFDS analysis to large-scale programs.

State of the Art. Several studies have been presented to reduce memory footprint and thus improve the scalability of IFDS solvers. Heros [17] and FlowDroid's FastSolver [2] reduce their memory footprints by constructing program representations in a demand-driven way. CleanDroid [5] adopts a semantic garbage collector for IFDS-based analyses to reclaim memory. Although the above attempts reduce memory footprint to some extent, they can only partially address the problem, especially when analyzing large-scale programs.

DiskDroid [4] is an IFDS-based taint analysis tool that extends FlowDroid with a disk-assisted IFDS solver. The disk-assisted approach improves the memory scalability of existing IFDS implementation by exporting/importing certain edges into/from disks. Unfortunately, DiskDroid heavily relies on a heuristic-based strategy to decide when and what data to swap between memory and disks. This strategy could cause frequent fine-grained I/O accesses, leading to poor performance. Even worse, as the swapping strategy is ad-hoc, it is still very likely to run out of memory when analyzing large-scale programs. Graspan [18], [19] is a disk-based system supporting context-sensitive static analyses via function inlining [20] rather than summary-based approach [1]. Graspan is designed for more general static analysis problems that can be formulated as CFL-reachability [21], [22], [23], of which the IFDS problems are only a subset. However, Graspan fails to propose an optimized design specific to the IFDS problems for the sake of generality. We discuss the experimental data showing the inefficiency of both DiskDroid and Graspan in Section IV.

Our Insight & Approach. The IFDS problems can be formulated as a special kind of graph-reachability problem [1]. The core computation of the graph-reachability is to constantly generate a large number of new transitive edges (termed as "*path-edges*") by joining[1] with a fixed set of edges in the

† Corresponding authors.

[1]Here "join" refers to an operation performed to establish a connection between two or more edges, which is similar to the "table join" in database terminology, rather than the "join transformer" in abstract interpretation.

exploded super-graph (Definition 2). According to the memory usage statistics reported by [4], the path-edges dominate the memory consumption in FlowDroid's FastSolver. In particular, the memory usage of the path-edges accounts for an average of 79.07% of the total memory consumed in their experiments, while that of other edges is relatively small. Analyzing large-scale programs could generate too many path-edges to fit into the memory of a commodity PC, resulting in poor scalability and even infeasibility. We observed that the IFDS algorithm demonstrates the *locality* to some extent, meaning that its core computation around a path-edge only relies on the path-edge and all the edges in the exploded super-graph and requires no other path edges. This observation means that it is unnecessary to maintain all the path-edges in memory all the time. On the contrary, to improve the scalability, we can store the path-edges on disks and process several edge chunks simultaneously by performing streaming-based parallel computation.

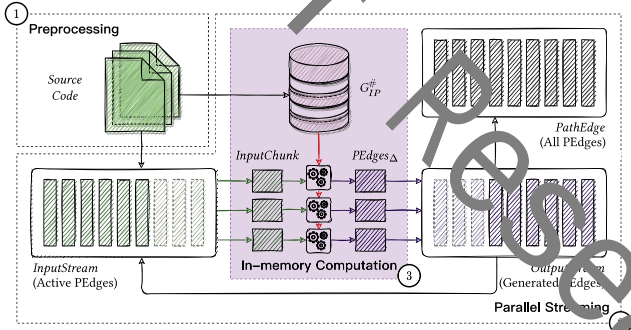


Fig. 1: The workflow of DStream

Based on the above insight, we devise a scalable IFDS framework named DStream, whose workflow is demonstrated as Figure 1. DStream consists of three crucial components: ① preprocessing, ② streaming, and ③ in-memory computation indicated with purple background. Here we briefly overview the overall workflow and elaborate on each component in detail in Sections III-A to III-C shortly.

Given a program to be analyzed, we first preprocess the program to construct the exploded super-graph and the initial set of path-edges (termed as *seed path-edges* or simply *seeds*) accordingly (cf. ①). We load the exploded super-graph $G^{\#}_{IP}$ into memory, followed by iterative streaming-based processing (cf. ②). At the beginning of each iteration, we partition the input-stream consisting of path-edges into multiple smaller chunks, which are loaded into memory and processed individually. Having each chunk, an in-memory computation phase (cf. ③) is performed to join the path-edges contained in the loaded chunk with the edges of $G^{\#}_{IP}$ based on the CFL-based formulation (Definition 3), and generate new path-edges ($PEdges_{\triangle}$) which will be stored to disks in a streaming way. At the end of each iteration, DStream checks whether the fixed point is reached. If not, the output-stream is used as the input-stream for the next iteration, and the computation continues.

The design of DStream is beneficial to both the scalability and efficiency of the IFDS framework. First, the maximum memory usage during computation is under control. We can readily control the maximum memory consumption by tuning the sizes of chunks and I/O buffers to fit the available memory. Second, once several chunks are loaded into memory, they can be processed data-parallelly to accelerate the reachability computation.

Contributions. We make the following contributions:

- We propose a systematic streaming-based out-of-core approach to significantly improve the memory scalability of the IFDS framework.
- We present a fine-grained data-parallel approach to effectively enhance the analysis parallelism of the IFDS framework.
- We designed and implemented a scalable IFDS framework named DStream, which is publicly available at https://github.com/DStream-project.
- We evaluated DStream over a comprehensive set of benchmarks. The experiments validated that DStream running on a commodity PC with limited available memory outperforms all other tools with average speedups from 4.37x to 14.46x. Meanwhile, the experiments confirmed that DStream successfully scales large-scale programs which FlowDroid and/or DiskDroid fail to analyze under the limited available memory.

Outline. The rest of this paper is structured as follows. We start with the necessary background about the IFDS framework in Section II. Section III outlines the design of our DStream framework, followed by an empirical evaluation of the DStream framework in Section IV. Finally, we survey related work in Section V and Section VI concludes.

II. Background

This section provides the necessary background about the IFDS framework, including the definition of the IFDS problem (§II-A), the IFDS algorithm (§II-B), and how the IFDS algorithm is formulated as a CFL-reachability problem (§II-C).

A. The IFDS Problem

In the classic IFDS formulation proposed by Reps *et al*. [1], a program is represented as a *super-graph* $G^* = (N^*, E^*)$ (aka. interprocedural CFG or simply ICFG). G^* is comprised of a set of CFGs G_0, G_1, …, and G_n (each for per method), one of which, G_0 indicates the entry method of the program. These intraprocedural CFGs are connected with *call-edges* and *return-edges* to construct the final G^*. For a method p, its CFG G_p has a unique *start-node* $s_p \in N^*$, and a unique *exit-node* $x_p \in N^*$. The other nodes represent the statements and predicates as usual, except that a callsite is represented by two nodes, a *call-node* $c \in N^*$ and a *return-node* $r \in N^*$ for convenience. E^* is a set of control-flow edges between nodes. There are four kinds of edges in E^*. The ordinary intraprocedural edges in G_p are termed *normal-edges*. For each callsite with its call-node c and return-node r, an intraprocedural *call-to-return-edge* connects c to r; an interprocedural *call-edge* connects c to the start-node s_q of its callee method q; and an interprocedural *return-edge* connect the exit-node x_q to r. Thus, dataflow facts can propagate interprocedurally via call-edges and return-edges.

Based on the definition of the super-graph, the IFDS problem can be defined as follows:

Definition 1 (**IFDS Problem**). An instance *IP* of an IFDS problem is a five-tuple $(G^*, D, F, M, \sqcap)$, where:

1) $G^* = (N^*, E^*)$ is a super-graph as defined above.
2) D is a finite set of dataflow facts.
3) $F \subseteq 2^D \mapsto 2^D$ is a set of distributive functions.
4) $M : E^* \mapsto F$ is a map from E^* to dataflow functions.
5) The meet operator $\sqcap$ is either union or intersection.

B. The IFDS Algorithm

To solve the IFDS problems precisely and efficiently, Reps *et al.* [1] proposed the IFDS algorithm, which transforms an IFDS problem into a graph reachability problem. The key is to represent the distributive transfer functions as graphs. Equivalently, each function can be represented as a bipartite graph with $2(D+1)$ nodes and at most $(D+1)^2$ edges. Each node represents an element of D, and an edge $d_1 \to d_2$ occurs in the graph if and only if $d_2 \in f(\{d_1\})$. In this way, the *super-graph* G^* of *IP* is extended to an *exploded super-graph* (*ESG*) $G^{\#}_{IP}$, which combines the ICFG and the dataflow functions.

Definition 2 (**Exploded Super-Graph**). Given an IFDS problem instance $IP = (G^*, D, F, M, \sqcap)$, the *exploded super-graph* for *IP*, denoted by $G^{\#}_{IP} = (N^{\#}, E^{\#})$, is defined as follows:

1) The node set $N^{\#} = N^* \times (D \cup \{\mathbf{0}\})$,
2) The edge set $E^{\#} = \{\langle m, d_1\rangle \to \langle n, d_2\rangle \mid (m,n) \in E^* \wedge d_2 \in M(m,n)(\{d_1\})\}$,

where $\mathbf{0}$ signifies an empty set of facts and $M(m,n) \in F$ is the flow function associated with the edge $(m,n) \in E^*$.

In the rest of the paper, the notations of nodes and edges are extended from super-graph to exploded super-graph. For example, $\langle n, d\rangle$ is a *call-node* in $G^{\#}_{IP}$ if n is a *call-node* in G^*, and $\langle n, d_1\rangle \to \langle m, d_2\rangle$ is a *call-edge* in $G^{\#}_{IP}$ if $n \to m$ is a *call-edge* in G^*.

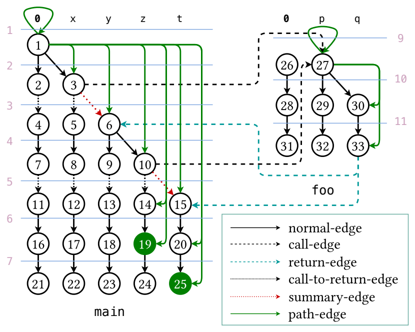
The IFDS algorithm is a worklist algorithm that takes $G^{\#}_{IP}$ as input for solving *IP*. Starting with the seeds, the algorithm maintains two sets during the computation:

- ***PathEdge*** records all the *path-edges*. A *path-edge* is a same-level realizable path of the form $\langle s_p, d_1\rangle \to \langle n, d_2\rangle$ which represents the suffix of a realizable path from $\langle s_{main}, \mathbf{0}\rangle$ to $\langle n, d_2\rangle$. A zero-length *path-edge* of the form $\langle s_p, d\rangle \to \langle s_p, d\rangle$ is termd as a *self path-edge*.
- ***SummaryEdge*** records all the *summary-edges*. A *summary-edge* is a same-level realizable path from $\langle n, d_1\rangle$ to $\langle m, d_2\rangle$, where n is a *call-node* and m is the matched *return-node*. A *summary-edge* summarizes interprocedural dataflow dependencies across method boundaries and can be reused at different callsites.

```
1   void main() {
2     x = source();
3     y = foo(x);
4     z = y;
5     t = foo(z);
6     sink(z);
7     sink(t);
8   }

9   Object foo(p) {
10    q = p;
11    return q;
12  }
```

(a) Example code



(b) Exploded super-graph

Fig. 2: The toy example of IFDS-based taint analysis. Figure 2a shows the example program consisting of two methods, `main` and `foo`. Figure 2b shows the corresponding exploded super-graph with *summary-edges* and a subset of *path-edges* related to node 19 and 25.

The algorithm accumulates *path-edges* and *summary-edges* until a fixed point is reached. Please refer to [1] for more technical details about the IFDS algorithm.

Example. Figure 2 gives a toy example showing how to apply the IFDS algorithm to taint analysis. Figure 2a shows the code of the example, where `main` is the entry method and `foo` is an identity method which is invoked by `main` twice. `source` and `sink` are the taint source method and the taint sink method, respectively. Figure 2b shows the corresponding exploded super-graph of the example. Each node in the exploded super-graph represents a tainted fact at a program point, and each edge represents the propagation of a tainted fact along the ICFG. For example, node 19 represents the fact that `z` is tainted at the program point after Line 6, and the path-edge $1 \to 1$ is a seed. Line 2 of `main` marks `x` as tainted, and one new *path-edge* $1 \to 3$ is generated. Line 3 calls `foo` with `x` as the argument and `y` as the return variable. The parameter `p` of `foo` is then marked as tainted along the *call-edge* $3 \to 27$, and thus one new *self path-edge* $27 \to 27$ is generated. As the analysis of `foo` finishes, `p` at the exit point of `foo` is marked as tainted, and it returns to `y` along the *return-edge* $33 \to 6$. Since the *call-edge* $3 \to 27$ and the *return-edge* $33 \to 6$ are matched, one new *summary-edge* $3 \to 6$ is generated. Therefore, the tainted fact is propagated along $3 \to 6$ and one new *path-edge* $1 \to 6$ is obtained. At the end of the computation, the *path-edges* $1 \to 19$ and $1 \to 25$ indicate that `z` and `t` are tainted at their sink points, respectively.

C. The CFL-reachability Formulation

Based on the IFDS algorithm, we can further formulate it as a context-free language- (CFL-) reachability problem.

Definition 3 (**CFL-reachability Formulation**). Given an IFDS problem instance *IP* with its exploded super-graph $G^{\#}_{IP}$, and the set of all the callsites of $G^{\#}_{IP}$ is denoted as *CallSite*. Let $\mathcal{L}$ be a context-free language over alphabet $\Sigma = \{e, n\} \cup \bigcup_{i \in CallSite}\{l_i, r_i\}$. Each path in $G^{\#}_{IP}$ defines a word over Σ obtained by concatenating the edge labels in order on the path. A path in $G^{\#}_{IP}$ is a $\mathcal{L}$-path if its word is a member of $\mathcal{L}$. We can formulate the IFDS algorithm as a CFL-reachability problem, in particular, a multi-source $\mathcal{L}$-path problem where $\mathcal{L}$ consists of the following productions:

$$N \leftarrow n \mid N\ e \mid N\ S \tag{1}$$
$$N \leftarrow N\ l_i \qquad (i \in \mathit{CallSite}) \tag{2}$$
$$S \leftarrow l_i\ N\ r_i \qquad (i \in \mathit{CallSite}) \tag{3}$$

Each (terminal or non-terminal) symbol represents a kind of edge: n represents a *seed path-edge*, e represents a *normal-edge* (or a *call-to-return-edge*), l_i and r_i represent a *call-edge* and a *return-edge* with context matching at the callsite i, N represents a *path-edge*, and S represents a *summary-edge*. The path-edge generated by production (2) is a self path-edge (*e.g.*, 27 $\rightarrow$ 27 in Figure 2). Productions (1) and (2) indicate the computation logic of path-edges, while production (3) indicates the computation logic of summary-edges. In the rest of the paper, we use the term "*original-edge*" to refer to an edge that has the type represented by the terminal symbols e, l_i, and r_i. In other words, all edges in $E^{\#}$ are *original-edges*.

The CFL-based formulation is equivalent to the original IFDS algorithm. Here we discuss the equivalence briefly. The initialization of ***PathEdge*** and ***WorkList*** in the IFDS algorithm corresponds to the production $N \leftarrow n$. There are 3 cases in the IFDS algorithm. The first case corresponds to the productions $N \leftarrow N\ l_i$ and $N \leftarrow N\ e \mid N\ S$, the second case corresponds to the productions $S \leftarrow l_i\ N\ r_i$ and $N \leftarrow N\ S$, and the third case corresponds to the production $N \leftarrow N\ e$.

The CFL-based formulation characterizes the core computation of the IFDS algorithm. Based on the formulation, the IFDS framework can be modeled as a dynamic transitive-closure computation over the *exploded super-graph* $G^{\#}_{IP}$. The essential operation of the dynamic transitive-closure computation is the join operation between edges. For example, given $1 \xrightarrow{N} 6$ and $6 \xrightarrow{e} 10$ in Figure 2, we can join them and obtain $1 \xrightarrow{N} 10$ according to $N \leftarrow N\ e$. For $S \leftarrow l_i\ N\ r_i$, an example is $\langle 3 \xrightarrow{l_0} 27, 27 \xrightarrow{N} 33, 33 \xrightarrow{r_0} 6\rangle$, we can obtain $3 \xrightarrow{S} 6$ after performing a double-join operation on them.

III. The DStream Framework

In this section, we describe the design and implementation of our streaming-based data-parallel IFDS framework, DStream, which aims to tackle the poor scalability problem of the IFDS framework. We have developed DStream as a *single-machine*, *out-of-core* system which enables precise IFDS analysis on a commodity PC with limited memory.

The IFDS algorithm essentially performs iterative reachability computation on an *exploded super-graph* until all the *path-edges* are generated [1]. Based on the CFL-formulation (Section II-C), the core computation around a path-edge is locality-aware, and it is not necessary to maintain all the path-edges in memory all the time during the computation. On the contrary, to improve the scalability, we can store ***PathEdge*** on disks and process a part of ***PathEdge*** at a time by performing streaming-based parallel computation. In this way, we improve the scalability of the IFDS framework from two aspects. First, we propose a systematic streaming-based out-of-core computation model to reduce memory footprint significantly. Second, we adopt fine-grained data-parallel computation, which can effectively improve computation efficiency.

Algorithm 1: Streaming-based IFDS Algorithm

```
   Input: $P$: program under analysis
   Data: $PEdges$, $SEdges$: all path-/summary-edges;
         $PEdges_\triangle$, $PEdges_{tmp}$: active/inactive new path-edges;
         $SEdges_\triangle$: new summary-edges
   /* Preprocessing */
1  $\langle G^{\#}_{IP}, InputStream\rangle \leftarrow$ PreProcess($P$)
   /* Streaming-based computation */
2  $SEdges \leftarrow \varnothing$
3  repeat
4      $OutputStream \leftarrow \varnothing$
5      $ListSEdges_\triangle \leftarrow \varnothing$
       /* Parallel streaming phase */
6      while hasNext($InputStream$) do in parallel
7          $InputChunk \leftarrow$ StreamIn($InputStream$)
           /* In-memory computation */
8          $\langle PEdges_\triangle, PEdges_{tmp}, SEdges_\triangle\rangle \leftarrow$
             InMemComputation($InputChunk, E^{\#}, SEdges$)
9          $OutputStream \leftarrow$ StreamOut($PEdges_\triangle$)
10         $PEdges \leftarrow$ StreamOut($PEdges_{tmp}$)
11         insert $SEdges_\triangle$ to $ListSEdges_\triangle$
       /* Synchronization phase */
12     $SEdges \leftarrow SEdges \cup ListSEdges_\triangle$
13     $PEdges \leftarrow PEdges \cup OutputStream$
14     $InputStream \leftarrow OutputStream$
15 until $InputStream = \varnothing$
```

Algorithm 1 presents a high-level streaming-based, data-parallel algorithm. The algorithm takes as input a program to be analyzed and preprocesses it for the subsequent computation (Line 1). The following loop is a streaming-based BSP-style [24] iterative process (Lines 3–15), where each iteration is termed as a *superstep*. Each superstep consists of two phases, namely the parallel streaming (Lines 6–11) and synchronization (Lines 12–14). The above process is repeated until the *InputStream* is empty. In the following, we elaborate on each crucial component of our algorithm, including preprocessing (§ III-A), streaming-based processing (§ III-B), and in-memory computation (§ III-C).

A. Preprocessing

Preprocessing prepares for the core computation. Given a program to be analyzed, DStream preprocesses it to generate an *exploded super-graph* $G^{\#}_{IP}$ for a specific IFDS problem instance *IP*, while building certain edge index and initializing the seed path-edges to facilitate the subsequent computation. The generated $G^{\#}_{IP}$ consists of a set of original-edges, whereas the node information is implicitly encoded in the edges, thus reducing memory overhead. An original-edge is represented by a triple $\langle src, tgt, label\rangle$ whose elements denote the *source-node*, the *target-node*, and the edge label (corresponding a terminal symbol in Definition 3), respectively. During preprocessing, all the original-edges are loaded into memory.

Edge Indexing. As mentioned earlier, the core IFDS computation of DStream is to perform the *join* operation between labeled edges. As intensive queries of edges would happen during joining, we must be able to quickly find needed edges for efficient join implementation. The indexing scheme for the original-edges is critical for good performance. Here we adopt a two-layered index for an efficient query of edges. The first layer indexes the graph by source-node, and the second one indexes the graph by edge label. For example, suppose we would like to perform the CFL-reachability computation based on the production $N \leftarrow N\ e$. Given a path-edge $x \xrightarrow{N} y$, to complete the computation, we need to acquire all the normal-edges (*i.e.*, e-edges) whose source-node is y. Based on the two-layered indexing, we utilize the first layer index to locate all the original-edges with y as their source-node, and then apply the second layer index over the above-indexed edges to identify all the normal-edges needed quickly. In this way, we can achieve effective indexing, thus supporting efficient in-memory CFL-reachability computation, especially double-join operations (§ III-C).

PathEdge Initialization at Multiple Entry Points. The original IFDS algorithm takes the seed path-edge $\langle s_{main}, \mathbf{0}\rangle \rightarrow \langle s_{main}, \mathbf{0}\rangle$ of *main* method as the initial *PathEdge*, where *main* is the only one entry point of the program. However, for some programs under analysis, there may be more than one entry point. For example, Android applications do not have a main method like the traditional Java programs. Instead, many entry points are implicitly called by the Android framework. Another example is the library code, where generally no entry method exists. But there may be sample code or test code that can be used as entry points for analysis. For such codebases, different entry points may lead to different components of a large-scale program. To cover as much code as possible during analysis, we collect all the entry points of a program to be analyzed and take all the seed path-edges (*i.e.*, n-edges) of these entry points as the initial *InputStream*. The initial *InputStream* is stored onto disks and used later to turn on the core computation of streaming-based processing (§ III-B).

B. *Streaming-based Processing*

After the preprocessing, the streaming-based processing is started (Lines 3–15 in Algorithm 1). Streaming-based computation essentially provides a scalable and efficient locality-aware graph reachability computation. The computation is represented as the *join* of two sets of edges where the small set of static edges are held in memory, and a large number of dynamic edges are streamed from/to disks. In the DStream framework, based on CFL-reachability discussed in § II-C, the two sets of edges to be joined are *original-edges* (plus *summary-edges*) and *path-edges* constantly generated during computation. As described in § III-A, we load the *original-edges* into memory and store the *path-edges* on disk before the streaming-based computation. The core of streaming-based computation is a BSP-style iterative process, where each iteration is termed as a *superstep*. Each superstep consists of two phases, namely the parallel streaming phase (Lines 6–11) and synchronization phase (Lines 12–14).

At the parallel streaming phase, DStream takes as input the *InputStream*, which is a file storing all the active path-edges and launches a parallel streaming process. At first, the *InputStream* is divided into several smaller chunks of (almost) fixed size according to available memory. DStream handles the chunks by multiple threads in parallel. In particular, each thread loads one chunk into memory. Next, each thread invokes the in-memory computation procedure InMemComputation (which will be elaborated shortly in § III-C) to generate new path-edges and summary-edges (Line 8). The newly generated path-edges are divided into two categories: active *PEdges*$_\triangle$ and inactive *PEdges*$_{tmp}$. Active *PEdges*$_\triangle$ is exported to the *OutputStream* for the next superstep (Line 9), while the processing of inactive *PEdges*$_{tmp}$ has been done in current superstep and is exported to result *PEdges* directly (Line 10). As the chunks are independent of each other, we can safely launch multiple threads to process the entire *InputStream* simultaneously. For the sake of performance, we maintain a separate buffer of *PEdges*$_\triangle$ and a file of *OutputStream* for each thread, thus avoiding synchronization costs. Apart from the original-edges, the computation of path-edges, as shown in Production 1, also demands the dynamically generated summary-edges. To ensure the computation's correctness, we also keep the summary-edges in memory. Maintaining summary-edges in memory has no significant impact on scalability since the number of summary-edges is much smaller compared with that of path-edges, and we will delete the summary edges that are no longer needed from memory during the computation (§ III-C).

Once the parallel streaming phase finishes, a synchronization phase is conducted. In particular, the in-memory summary-edges are updated by combining the newly generated ones (Line 12). The *OutputStream* is aggregated to *PEdges* (Line 13), and the *OutputStream* is treated as the new *InputStream* for the next superstep (Line 14). After synchronization, the next superstep is scheduled until the *InputStream* is empty.

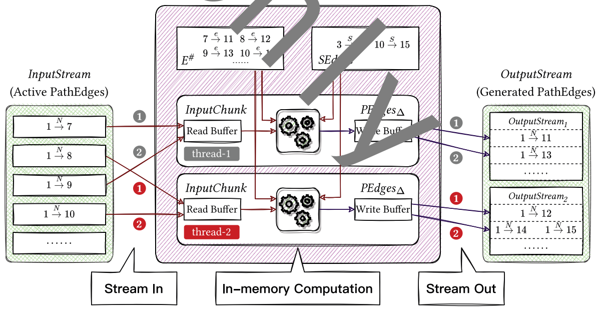


Fig. 3: Streaming-based data-parallel computation

Example. Figure 3 illustrates the process of parallel streaming for the running example (Figure 2). For simplicity, we assume only two threads (thread-1 and thread-2) are available to illustrate the main idea of the process. We first divide the *InputStream* into smaller chunks as described earlier. In Figure 3, (more than) 5 chunks are divided, each containing one path-edge. We take the first four chunks as an example to demonstrate the streaming process. At the very beginning, thread-1 and thread-2 are both idle. Each can thus be assigned to load and process one chunk in parallel. For example, thread-1 loads chunk-1 ($1 \xrightarrow{N} 7$) into its local read-buffer, while thread-2 reads chunk-2 ($1 \xrightarrow{N} 8$) into its local buffer. The two threads then perform the in-memory computation independently by joining the path-edges in their local buffers with the globally shared original-edges and summary-edges. For brevity, Figure 3 shows a simplified version of in-memory computation (§ III-C), which processes only the path-edges in the *InputStream* and then outputs them directly to the *OutputStream*. In particular, thread-2 joins $1 \xrightarrow{N} 7$ in chunk-1 with $7 \xrightarrow{N} 11$ and generates one new path-edge $1 \xrightarrow{N} 11$. Simultaneously, thread-2 generates one new path-edge $1 \xrightarrow{N} 12$. The path-edges newly generated by each thread are stored in their local write-buffers. Once the write-buffer is full (or the chunk has been processed completely), we flush the write-buffer to *OutputStream* which is stored on disks. Here each thread associates with one separate *OutputStream* file on disks. As shown in Figure 3, thread-1 exports the new path-edge $1 \xrightarrow{N} 11$ to its associated file *OutputStream*$_1$. Similarly, thread-2 writes the path-edge generated $1 \xrightarrow{N} 12$ to *OutputStream*$_2$. Once a thread finishes processing one chunk, we check whether there are still unprocessed chunks. If so, the idle thread will load and process the next chunk as described above. In Figure 3, thread-1 finishes the processing of chunk-1 and then streams in chunk-3; the processing is similar to the above. The process continues until all the chunks in *InputStream* have been processed.

C. In-memory Computation

This section presents the details of in-memory computation, as well as the designs for improving efficiency. During in-memory computation, DStream performs the core computation of the IFDS algorithm according to the CFL-reachability formulation. The essence of the CFL-reachability formulation is to generate new reachable edges by joining labeled edges. Algorithm 2 shows the details of the in-memory computation. It takes as inputs an *InputChunk* loaded, the original-edges $E^\#$, and the *SEdges*. At the beginning of the computation, we initialize the sets of active new path-edges $PEdges_\triangle$, inactive new path-edges $PEdges_{tmp}$, and new summary-edges $SEdges_\triangle$ as empty (Line 1), and then process each of the active path-edges in *InputChunk* one by one (Lines 3–22) to generate new path-edges (and summary-edges). The newly generated $PEdges_\triangle$ will be used as (part of) the input for the next superstep. At the end of Algorithm 2, all the new path-edges ($PEdges_\triangle$ and $PEdges_{tmp}$) and the new summary-edges $SEdges_\triangle$ are returned.

Correctness and Termination. For each path-edge $x \xrightarrow{N} y$ to be processed, we perform its CFL-reachability computation

Algorithm 2: InMemComputation

```
    Input: $InputChunk$, $E^\#$, $SEdges$
1   $PEdges_\triangle, PEdges_{tmp}, SEdges_\triangle \leftarrow \varnothing$
2   $Worklist \leftarrow InputChunk$
3   while $Worklist \neq \varnothing$ do
4       pop $x \xrightarrow{N} y$ from $Worklist$
        /* case 1: $N \leftarrow N\ e$ */
5       foreach $y \xrightarrow{e} z \in E^\#$ do
6           if $z$ is a join- or call-node then
7               if $x \notin src(z)$ then  add $x$ to $src(z)$
8               else  continue
9           add $x \xrightarrow{N} z$ to $PEdges_{tmp}$ and $WorkList$
        /* case 2: $N \leftarrow N\ S$ */
10      foreach $y \xrightarrow{S} z \in SEdges$ do      // $z$ is return-node
11          if $x \notin src(z)$ then
12              add $x$ to $src(z)$
13              add $x \xrightarrow{N} z$ to $PEdges_{tmp}$ and $WorkList$
        /* case 3: $N \leftarrow N\ l_i$ */
14      foreach $y \xrightarrow{l_i} z \in E^\#$ do      // $z$ is start-node
15          if $init(z).isFalse()$ then
16              add $z \xrightarrow{N} z$ to $PEdges_\triangle$
17              $init(z).setTrue()$
        /* case 4: $S \leftarrow l_i\ N\ r_i$ and $N \leftarrow N\ S$ */
18      foreach matched $x \xrightarrow{l_i} t$ and $y \xrightarrow{r_i} z \in E^\#$ do
19          add $t \xrightarrow{S} z$ to $SEdges_\triangle$
20          foreach $a \in src(t)$ do      // $t$ is call-node
21              add $a$ to $src(z)$      // $z$ is return-node
22              add $a \xrightarrow{N} z$ to $PEdges_{tmp}$ and $WorkList$
23  return $\langle PEdges_\triangle, PEdges_{tmp}, SEdges_\triangle \rangle$
```

based on the productions as shown in Definition 3. The first case (Lines 5–9) corresponds to $N \leftarrow N\ e$. The second (Lines 10–13) and third (Lines 14–17) cases correspond to $N \leftarrow N\ S$ and $N \leftarrow N\ l_i$, respectively. The fourth case (Lines 18–22) corresponds to $S \leftarrow l_i\ N\ r_i$ and $N \leftarrow N\ S$, which actually performs a double-join operation and necessary joining between N edges and the generated S-edge. With the correspondence between Algorithm 2 and the CFL-reachability formulation (Definition 3), we can immediately obtain the correctness of Algorithm 2.

The termination of Algorithm 1 is determined by the emptiness of *InputStream*, which indicates that no new active path-edges are generated in the current superstep. However, an edge may be generated in multiple ways. For instance, both edge pairs $x \xrightarrow{N} a \xrightarrow{e} y$ and $x \xrightarrow{N} b \xrightarrow{e} y$ could generate the same edge $x \xrightarrow{N} y$ and propagate further to generate more duplicated edges. These duplicated edges may cause the computation to fail to terminate. Therefore, the key to termination is to avoid duplication of path-edges. One source of duplication is *join-node*, which has multiple incoming intraprocedural original-edges. A return-node can also become a join-node during the computation as the summary-edges are generated. We thus consider all the return-nodes as potential join-nodes. To avoid duplication of path-edges, we maintain a reverse index src

of the path-edges for each join-node. With src, no duplicated path-edges will be generated and propagated further as shown in Lines 7 and 11. Another source of duplication is *start-node*, which has multiple incoming interprocedural call-edges. As shown by Algorithm 2, only non-initialized self path-edges are considered active path-edges at the beginning. Therefore, we only need to take care of the duplication of self path-edges. For a start-node x, we synchronize on $init(x)$ to ensure that only one self path-edge can be generated (and therefore only in one thread). No duplicated self path-edges will be processed in two different threads in the next superstep.

Computation Closure. Naively, all the new path-edges generated according to the edges of the *InputChunk* should be put into *PEdges*$_\triangle$ which becomes the input for the next superstep (as shown by the naive version in Figure 3). However, a new path-edge could be processed immediately to generate more transitive path-edges in a superstep, rather than exporting to *OutputStream* and being processed in the next superstep. In this way, each thread can produce as many reachable path-edges as possible to accelerate the convergence and thus reduce the I/O costs. In particular, we maintain *WorkList* for the closure computation and initialize it with *InputStream* (Line 2). *PEdges*$_{tmp}$ contains the new path-edges processed in the current superstep. All new non-self path-edges are added to *WorkList* as well as *PEdges*$_{tmp}$ and processed immediately, and all new self edges are added to *PEdges*$_\triangle$. The in-memory computation continues until all the path-edges in *WorkList* have been processed. Note that although such full closure computation brings performance benefits, it could lead to massive memory consumption, which smashes the scalability. Fortunately, we can do fine-grained tuning to terminate the closure computation at any moment and stream all the unprocessed path-edges to *OutputStream*, which would not affect the correctness.

Double-join Implementation. In the CFL-reachability formulation shown as Definition 3, the production $S \leftarrow l_i\ N\ r_i$ involves three symbols, which means that we need to perform a double-join operation on three edges labeled as the RHS to generate a summary-edge. Given the double-joining of a path-edge $x \xrightarrow{N} y$, not only all the return-edges $y \xrightarrow{r_i} \blacktriangle$ but also the call-edges $\blacktriangledown \xrightarrow{l_i} x$ are required. All the return-edges $y \xrightarrow{r_i} \blacktriangle$ can be found in an efficient way by indexing on the source node y and label r_i based on the two-layered index structure. However, it can not support the efficient query of call-edges $\blacktriangledown \xrightarrow{l_i} x$ since the source node is unknown. The key to efficient double-joining is quickly finding all the call-edges whose target node is x. To this end, we adopt a new approach by introducing a reverse edge $x \xrightarrow{\overline{l_i}} \blacktriangledown$ for each call-edge $\blacktriangledown \xrightarrow{l_i} x$ in the exploded super-graph. In this way, we can obtain all the reverse call-edges via the edge index and perform the double-joining efficiently.

Edge Deletion. DStream keeps all the original-edges in memory throughout the computation. Although DStream maintains these edges in a very compact way, they still consume considerable memory for a large-scale graph. We observed that many normal-edges are not used all the time. For instance, a normal-edge having only one predecessor path-edge only needs to be used once; they are redundant after their use. As such, we can delete redundant original-edges from memory in time to reduce memory footprint. We propose a simple but effective edge deletion strategy: given a normal-edge, if all its predecessor normal-edges have been deleted, this edge can also be deleted safely. The simplest case is the normal-edge whose source-node is a start-node. According to the IFDS algorithm, each start-node will be processed only once to produce a self path-edge. Therefore, the normal-edges whose source-node is a start-node will also be used only once and can be deleted after the use. The summary-edges can be deleted according to the same strategy. By deleting edges, we can reclaim memory so that more path-edges can be processed in a subsequent superstep, which in turn accelerates the convergence of computation.

Example. Figure 4 shows a complete example to illustrate how computation closure, double-join, and edge deletion work, where the computation reaches the fixed point after three supersteps. We use only one thread here for brevity, as the parallel computation has been illustrated in Figure 3. At the very beginning, Figure 4a shows the exploded super-graph and the seed path-edge $\{N_0\}$. Figure 4b shows the first superstep, whose input-stream is $\{N_0\}$. In this superstep, DStream computes the closure for N_0 as $\{N_i \mid i \in [1,4]\}$ and generates one new self path-edge N_5 which is put into *PEdges*$_\triangle$. Note that node 2 is a call-node; thus, its associated src is updated to $\{0\}$. The normal-edges $e_0{\sim}e_3$ are deleted at the end of the superstep. Figure 4c shows the second superstep, whose input-stream is $\{N_5\}$. In this superstep, two path-edges N_6 and N_7 are generated first, and then the summary-edge $S_0 : 2 \xrightarrow{S} 5$ is also generated by the double-joining performed between $\overline{l_0}$, N_7 and r_0. As $src(2)$ is $\{0\}$, then N_8 is propagated through S_0, $src(5)$ is updated to $\{0\}$, and N_9 is generated as part the computation closure of N_8. The normal-edges $e_4{\sim}e_6$ are deleted at the end of superstep 2. Note that S_0 generated in this superstep is also deleted as there is no predecessor edge for node 2. For the third superstep shown in Figure 4d, whose input-stream is empty, there is no more path-edge to be processed, and the computation terminates.

IV. Evaluation

To demonstrate the performance and scalability of our streaming-based data-parallel DStream framework, we conducted a comprehensive set of experiments. Since the precision and effectiveness of the IFDS algorithm have already been validated in prior work [2], [18], we mainly focus on the efficiency and scalability of DStream in our evaluations and discuss analysis precision briefly in § IV-E. Our evaluation seeks to answer the following three research questions:

RQ1. How does DStream perform and how does it compare with state-of-the-art tools?

RQ2. How about the thread scalability of DStream?

RQ3. How about the memory scalability of DStream?

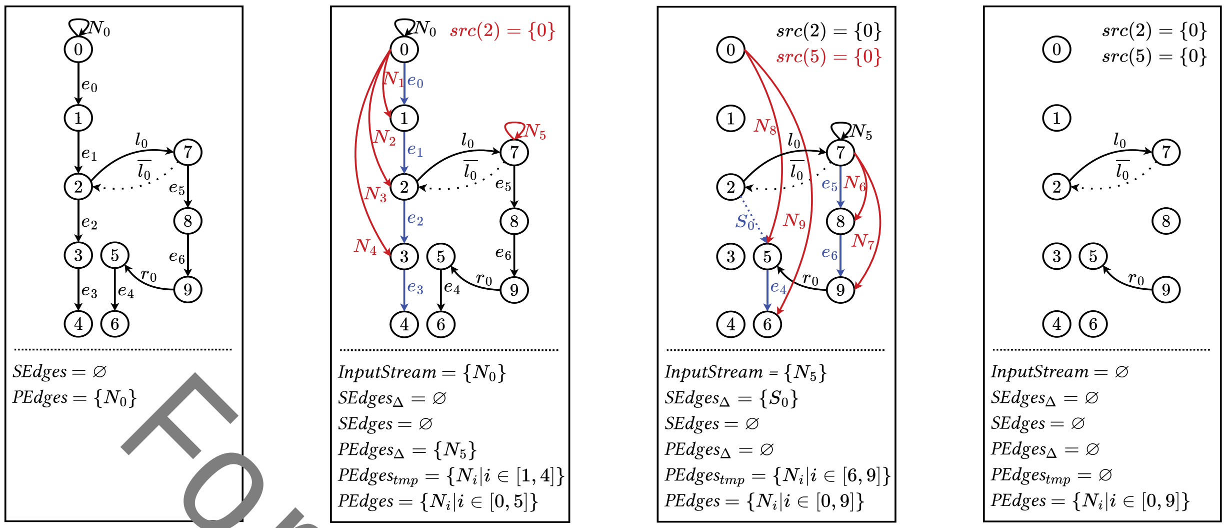

N_0
e_0
e_1
e_2
e_3
e_4
e_5
e_6
l_0
$\bar{l}_0$
r_0
$SEdges = \varnothing$
$PEdges = \{N_0\}$
$src(2) = \{0\}$
N_1
N_2
N_3
N_4
N_5
$InputStream = \{N_0\}$
$SEdges_\Delta = \varnothing$
$SEdges = \varnothing$
$PEdges_\Delta = \{N_5\}$
$PEdges_{tmp} = \{N_i | i \in [1, 4]\}$
$PEdges = \{N_i | i \in [0, 5]\}$
$src(5) = \{0\}$
N_6
N_7
N_8
N_9
S_0
$InputStream = \{N_5\}$
$SEdges_\Delta = \{S_0\}$
$SEdges = \varnothing$
$PEdges_\Delta = \varnothing$
$PEdges_{tmp} = \{N_i | i \in [6, 9]\}$
$PEdges = \{N_i | i \in [0, 9]\}$
$InputStream = \varnothing$
$SEdges_\Delta = \varnothing$
$SEdges = \varnothing$
$PEdges_\Delta = \varnothing$
$PEdges_{tmp} = \varnothing$
$PEdges = \{N_i | i \in [0, 9]\}$
(a) Input
(b) Superstep-1
(c) Superstep-2
(d) Superstep-3

Fig. 4: An example to illustrate the process of double join, edge deletion, and computation closure. The red edges are newly generated in each superstep, and the blue edges will be deleted at the end of the superstep.

TABLE I: Characteristics and performance data of taint analysis on 36 subject apps. The first 3 columns provide the essential information about the apps. The middle 3 columns depict the complexity of analyzing these apps. The last 4 columns report the analysis time (in seconds) of each tool with 8 threads, 16 GB maximum available memory, and the time limit of 1 hour. OOM indicates out of memory error, while OOT indicates out of the time limit.

App.	Abbr.	Version	Size	#Source	#Sink	FlowDroid	DiskDroid	Graspan	DStream
F-Droid	FDD	1.1	7.[illegible]MB	92	380	756	739	399	103
acr.browser.lightning	ABL	5.1.0	[illegible].3MB	23	110	OOM	OOM$_{gc}$+OOT	530	126
bus.chio.wishmaster	BCW	1.0.2	3.[illegible]MB	28	313	212	180	105	24
com.alfray.timeriffic	CAT	1.09.05	347KB	[illegible]6	117	200	512	90	14
com.app.Zensuren	CAZ	1.21	177KB	93	13	9	11	4	3
com.genonbeta.TrebleShot	CGT	1.4.2	4.2MB	19	379	OOM	OOT	563	94
com.github.axet.bookreader	CGAB	1.12.14	28MB	[illegible]	[illegible]01	OOM	725	194	50
com.github.axet.callrecorder	CGAC	1.7.13	5.6MB	[illegible]6	11[illegible]	OOM	593	195	30
com.ichi2.anki	CIA	2.9.4	11MB	65	[illegible]65	OOM	OOT	555	91
com.igisw.openmoneybox	CIO	3.4.1.11	10MB	42	213	OOM	OOT	554	102
com.ilm.sandwich	CIS	2.2.4f	3.0MB	6	7[illegible]	10	19	6	4
com.kanedias.vanilla.metadata	CKVM	1.0.4	6.3MB	4	18	55	399	13	9
com.kunzisoft.keepass.libre	CKKL	3.0.2	9.9MB	13	570	OOM	OOT	440	75
com.orgzly	COR	1.8.5	4.9MB	27	487	OOM	1288	400	86
com.poupa.vinylmusicplayer	CPV	1.3.0	6.2MB	32	298	16	13	4	4
com.zeapo.pwdstore	CZP	1.3.3	4.3MB	12	186	OOM	OOT	510	76
de.k3b.android.androFotoFinder	DKAA	0.8.0	1.4MB	32	394	OOM	OOT	573	93
dk.jens.backup	DJB	0.3.4	6.2MB	5	134	8	12	4	3
fr.gouv.etalab.mastodon	FGEM	2.28.1	28MB	85	287	OOM	OOT	457	107
hashengineering.groestlcoin.wallet	HGW	7.11.1	3.2MB	4	192	OOM	57[illegible]	189	68
im.vector.app	IVA	1.2.2	109MB	13	340	OOM	OOM$_{gc}$+OOT	612	156
nodomain.freeyourgadget.gadgetbridge	NFG	0.6.0	6.4MB	72	739	OOM	OOM$_{gc}$+OOT	638	131
nya.miku.wishmaster	NMW	1.5.0	3.4MB	19	282	198	210	84	31
org.adw.launcher	OAL	1.3.6	1.1MB	13	136	27	185	10	6
org.csploit.android	OCA	1.6.5	3.5MB	7	110	10	12	5	3
org.decsync.sparss.floss	ODSF	1.16.0-1	2.8MB	36	252	OOM	OOT	466	72
org.fdroid.fdroid	OFF	1.8	7.6MB	102	393	674	634	215	69
org.gateshipone.odyssey	OGO	1.2.0	3.5MB	23	230	OOM	OOT	641	113
org.kde.kdeconnect_tp	OKKT	1.13.5	4.4MB	10	225	OOM	861	284	39
org.lumicall.android	OLA	1.13.1	5.6MB	38	216	OOM	477	171	38
org.openpetfoodfacts.scanner	OOS	3.6.8	12MB	57	393	OOM	OOT	421	82
org.secuso.privacyfriendlyactivitytracker	OSPA	2.4	4.6MB	14	290	OOM	OOT	768	93
org.secuso.privacyfriendlytodolist	OSPT	2.1	2.3MB	8	50	43	80	19	4
org.secuso.privacyfriendlyweather	OSPW	2.1.1	4.8MB	50	35	207	243	71	16
org.smssecure.smssecure	OSS	0.16.12	13MB	67	273	OOM	OOT	610	110
org.yaxim.androidclient	OYA	0.9.3	1.9MB	45	217	174	241	62	17

A. Experimental Setup

Platform. All experiments were conducted on a commodity PC, with an 8-core 3.60GHz Intel i7-9700K CPU, 32 GB available memory, and 1 TB magnetic disk storage, running Ubuntu 20.04.3 LTS (Focal Fossa). In the following experiments, we set the maximum number of available threads of all tools to 8 (except for **RQ2**) and limited the maximum memory usage of all tools to 16 GB (except for **RQ3**).

Instance Analysis. Analogous to the declarative program analysis [25], [26], we separate the computation back-end from the client analysis implementations. DStream is a back-end analysis engine that supports various IFDS instance analyses. To implement an instance analysis in DStream, we need a front-end to generate input ESGs for DStream back-end. In our evaluation, we chose the taint analysis, the most common instance of the IFDS framework, as the instance analysis. We implemented a taint analysis front-end with Soot [27] and SPARK [28]. Instead of computing the aliased access paths incrementally, we leveraged the off-the-shelf SPARK alias analysis to construct the exploded super-graph, including all the potential aliased paths beforehand. We applied the same way as FlowDroid to parse and identify the entry points in Android apps. Therefore, the entry points considered by DStream are consistent with that by FlowDroid. We also adopted the same configuration files (*e.g.*, SourcesAndSinks.txt) that FlowDroid uses to find sources and sinks in Android apps.

Baselines. We selected three state-of-the-art tools: (memory-only) FlowDroid [2], (disk-assisted) DiskDroid [4] and (disk-based) Graspan [18] as the baselines, and compared them with DStream in terms of both efficiency and scalability. FlowDroid and DiskDroid themselves are taint analyzers. We can compare directly with them. We obtained the executables (jar files) of both DiskDroid and FlowDroid from the link for artifacts[2] of DiskDroid, and run DiskDroid with its default disk-swapping configuration (*i.e.*, group by source). Graspan is a general analysis framework that does not have a front-end for taint analysis. We ported the taint analysis front-end implemented for DStream to Graspan by adding function inlining. We obtained the executable of Graspan by building it from the source code, which is publicly available at GitHub[3].

Benchmarks. We adopted all the *open-source* apps from the benchmarks of [3] and [4] to form the benchmarks of our evaluation. Since we are focusing on analyzing large-scale programs, if an app can be analyzed by every tool (DStream and all the baselines) within 10 seconds, we filtered it out from the benchmarks. Columns 1–6 of Table I show the characteristics of the remaining 36 apps of the benchmarks in detail. The first 3 columns provide the essential information of the apps. The following 3 columns depict the complexity of analyzing these apps, where the app size approximates the size of the codebase of the app, and the numbers of the taint sources and sinks capture the taint analysis workloads of the app to some extent. We run each tool 5 times on each app and report the average data of 5 runs.

[2]DiskDroid artifacts: https://doi.org/10.6084/m9.figshare.13246316

[3]Graspan at GitHub: https://github.com/Graspan/graspan-cpp

B. DStream Performance

To understand the performance of DStream, we compared it with three state-of-the-art tools using taint analysis. We limited the maximum available memory to 16 GB for each tool, as our goal is to enable developers to benefit from precise static analysis on their development machines. Columns 7–10 of Table I show the analysis time (in seconds) for each tool. Note that the analysis time of DStream and Graspan (*i.e.*, the last two columns in Table I) include the time cost for constructing the input exploded super-graph with alias analysis. OOM indicates that the tool runs out of memory when analyzing the app, while OOT denotes that the tool can not finish the analysis of the app within 1 hour.

As shown in the last column of Table I, DStream can finish the analysis on most apps (28 of 36) within 100 seconds. Comparing the last two columns in Table I, we can see that DStream outperforms Graspan on all the apps (except they tie on the "small" app **CPV**), with a maximum speedup of 8.26x (**OSPA**) and an average speedup of 4.37x. The comparison reveals that DStream is much more efficient than Graspan, which is attributed to the fact that DStream adopts the streaming-based approach and is much less computing intensive than Graspan because of no function inlining.

Column 7 of Table I shows the results of FlowDroid. More than half of the apps (21 of 36) are marked as OOM, which indicates that FlowDroid runs out of memory on these apps. The fact confirms that the memory-intensive nature of the IFDS framework is a significant obstacle to its scalability. For the apps that FlowDroid can finish, DStream achieves a maximum speedup of 14.29x (**CAT**) and an average speedup of 7.11x.

Column 8 of Table I shows the analysis time of DiskDroid running on the benchmarks, in which almost half of the apps (15 of 36) are marked as OOT and 3 apps are marked as OOM$_{gc}$+OOT[4]. As DiskDroid is a disk-assisted IFDS solution with disk-swapping equipped, it should not run OOM in theory. The reason for DiskDroid running OOM is that the GC overhead limit is exceeded. More specifically, the error occurs when the JVM spends too much time performing GC and can only reclaim very little heap space. OOM$_{gc}$ shows that DiskDroid's ad-hoc disk-memory swapping strategy is ineffective for certain cases. For 15 apps that DiskDroid runs OOT, DiskDroid seems to be trapped into the prohibitively expensive fine-grained disk I/O. Even if we extended the time limit to 3 hours, DiskDroid still runs OOT on these apps. For 21 apps that at least one of FlowDroid and DiskDroid can finish, DiskDroid did not show absolutely better performance than FlowDroid. DiskDroid performed worse than FlowDroid on more than half of the apps (11 of 21), which suggests that the performance cost of DiskDroid's disk-swapping strategy

[4]FlowDroid/DiskDroid does not terminate immediately when OOT happens. New tasks are not allowed to submit. However, all submitted tasks continue to be processed, which may lead to OOM.

outweighs its memory gain. On all the apps, DStream consistently outperforms DiskDroid, with a maximum speedup of 44.33x (**CKVM**) and an average speedup of 14.46x.

The answer to RQ1: DStream finishes the analysis on most apps within 100 seconds with 8 threads and 16 GB maximum available memory and outperforms all other tools on all the apps. Overall, these results indicate that the streaming-based approach of DStream significantly improves the IFDS framework's performance in terms of analysis time and memory usage.

C. Thread Scalability

Parallelism support is crucial to fully utilize modern computing resources. To investigate the parallelism of DStream and answer **RQ2**, we run FlowDroid, DiskDroid, Graspan, and DStream on each subject app with 1, 2, 4, and 8 threads, respectively. In this experiment, we limited the maximum available memory to 16 GB and set the time limit to 1 hour for all tools.

Figure 5 shows the analysis time (in seconds) of DStream and the baselines on the benchmarks with varying numbers of threads. For the sake of brevity of the line charts, we have selected 4 apps (**FDD**, **OFF**, **DJB**, and **CAZ**) for presentation. These 4 apps are selected because, first, they can be analyzed by all the tools. Second, among these 4 apps, **FDD** and **OFF** cost the longest execution time of FlowDroid. **DJB** and **CAZ** cost the shortest execution time of FlowDroid. Thus, we use them as representative complex and simple cases, respectively.

We can see from Figure 5 that both DStream and Graspan scale almost linearly as the number of threads increases, no matter on complex or simple apps. However, it is surprising that their parallelism is far from satisfactory for the other two tools, especially on complex apps. For **FDD**, both FlowDroid and DiskDroid slow down as the number of threads increases from 2 to 8. For **OFF**, FlowDroid slows down as the number of threads increases from 2 to 8, while DiskDroid slows down as the number of threads increases from 4 to 8. The speedup trends of these tools differ so much because of the use of different parallelism mechanisms. FlowDroid/DiskDroid uses naive task parallelism, while DStream, as well as Graspan, devises a data-parallel algorithm by revisiting the CFL-reachability from a data-parallel perspective and solving it as a big-data problem. Data parallelism scales the computation by decomposing the data set into concurrent processing streams, all performing the same set of operations. Theoretically, the more threads a data-parallel algorithm has, the higher the speedup (before reaching the limit of the speedup).

The answer to RQ2: DStream, as well as Graspan, scales almost linearly as the number of threads increases, while FlowDroid and DiskDroid even slow down as the number of threads increases. It shows that fine-grained data-parallel computation is of significant importance for a highly parallel IFDS solver.

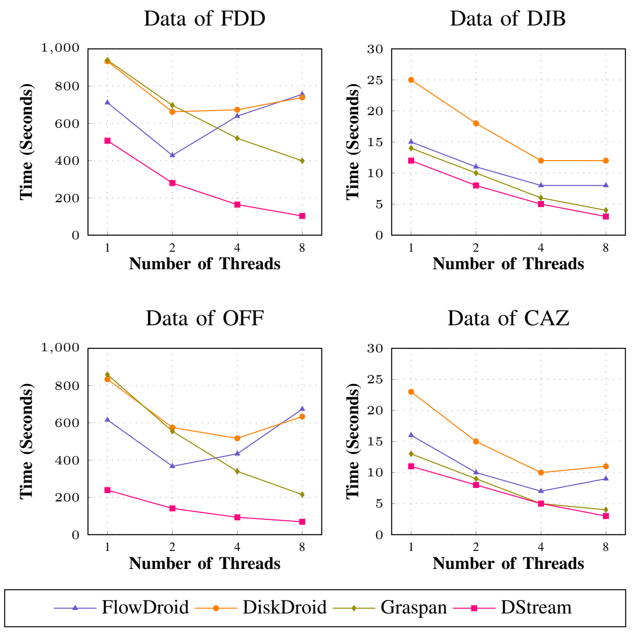


Fig. 5: Analysis time (in seconds) of 4 tools on 4 selected apps (**FDD**, **DJB**, **IFF** and **CAZ**) with varying numbers of threads (1, 2, 4, and 8).

D. Memory Scalability

The goal of DStream is to enable developers to leverage precise static analysis (in particular, the IFDS analysis) more practically on their development machines with limited available memory. To understand the memory scalability of DStream, we compared it with FlowDroid, DiskDroid, and Graspan using taint analysis with varying memory budgets. As the typical RAM configuration for a development machine does not exceed 32 GB typically (*e.g.*, the machine of our experimental platform), we conducted the experiment with different memory budgets (in GB) of 8, 16 and 32, respectively. In this experiment, we set the maximum number of threads to 8 and the time limit to 1 hour for all tools.

TABLE II: #OOM, #OOT and #Finished of apps for the 4 tools with varying memory budgets (in GB) of 8, 16, and 32. Note the sum of these numbers in each column of DiskDroid may not equal the number of all the apps of the benchmarks (*i.e.*, 36), as OOM+OOT could occur as shown in Table I.

	FlowDroid			DiskDroid			Graspan			DStream		
Mem (GB)	8	16	32	8	16	32	8	16	32	8	16	32
#OOM	28	21	18	3	3	2	0	0	0	0	0	0
#OOT	0	0	1	15	15	13	0	0	0	0	0	0
#Finished	8	15	17	21	21	23	36	36	36	36	36	36

Table II shows the numbers of (OOM, OOT and finished) apps for all tools with varying memory budgets. As shown, DStream and Graspan finished the analysis on all the apps with all different memory budgets. For FlowDroid, the number of finished apps increases with the memory budget, but more than half of the benchmarks (19 of 36) still failed to finish the analysis with 32 GB memory budget. Note that there is an

OOT app for FlowDroid at a 32 GB memory budget, but it does not appear at smaller memory budgets. The reason is that the app indeed took a very long time to finish the analysis at all memory budgets, but the analysis on the app at a smaller memory budget was terminated earlier by an out-of-memory error, which caused the app to be reported as OOM instead of OOT. The report of FlowDroid confirms again that the IFDS analysis is memory- and computing-intensive. For DiskDroid, the number of finished apps is consistently greater than that of FlowDroid, but the numbers of (OOM, OOT and finished) apps remain almost constant as the memory budget increases. The almost constant trend indicates that the ad-hoc disk-swapping strategy does not solve the memory bottleneck problem of the IFDS framework in general.

The answer to RQ3. DStream finishes the analysis on all the apps with varying memory budgets (in GB) of 8, 16, and 32. These results empirically show that DStream can achieve the goal of enabling developers to leverage the IFDS analysis on their development machines.

E. Analysis Precision

DStream theoretically has the same analysis precision as FlowDroid because the basic algorithm of taint analysis followed by DStream remains unchanged. In our experiments, we checked the analysis results reported by DStream and FlowDroid. DStream can identify all the source-sink pairs that FlowDroid/DiskDroid reports. Due to the discrepancy in the implementation of DStream and FlowDroid, the false positives they reported are slightly inconsistent. But they do not vary significantly regarding the number of false positives.

Note that compared with the demand-driven alias analysis used by FlowDroid, DStream adopts a less precise alias analysis (*i.e.*, SPARK [28]) to construct the exploded super-graph. But our experimental results validate that DStream can identify all the source-sinks pairs FlowDroid finds. The extra number of false positives reported by DStream is also negligible. It can be explained from two aspects. First, the flow-sensitivity seems not crucial for Java applications, and SPARK already obtains most of the benefit of flow-sensitivity by splitting variables; Second, both context- and flow-sensitivity are, to some extent, covered by the subsequent taint tracking. In other words, the false positives introduced by imprecise alias analysis are very likely to be filtered by the context- and flow-sensitive taint analysis afterward.

V. Related Work

In this section, we survey several work related to the IFDS framework[1] from the following aspects:

***Applicability*.** The IDE framework [29] generalizes the IFDS framework for the IDE problems, and more general CFL-reachability problems are described in [30]. Naeem *et al.* [31] provides several practical IFDS extensions, making it applicable to a wider class of interprocedural dataflow problems. Access-Path Abstraction [32] is a novel IFDS extension, which combines efficiency with maximal precision. Schubert *et al.* [33] builds an extendable IFDS/IDE solver for C/C++ in LLVM. Madsen *et al.* [34] formulates IFDS/IDE in Flix, a Datalog-inspired declarative programming language.

***Efficiency*.** Rodriguez *et al.* [13] introduces an actor-based concurrent IFDS implementation. Bodden [17] develops a generic (multi-threaded) implementation of IFDS/IDE solver in Soot [27]. Reviser [12] is an incremental IFDS algorithm. He *et al.* [3] provided a sparse IFDS algorithm to accelerate the performance. Shi *et al.* [14] presents a parallel bottom-up IFDS/IDE implementation that pipelines the sub-tasks.

***Scalability*.** WALA [35] offers a memory-efficient bit-vector-based IFDS implementation. Weiss *et al.* [36] proposed a database-backed strategy for IFDS problems. CleanDroid [5] is an IFDS implementation with semantic garbage collectors. DiskDroid [4] improves memory scalability through an ad-hoc disk-swapping strategy. Following the line of systemizing program analysis, various systems are developed to support scalable interprocedural analysis. Graspan [19], BigSpa [37], and Grapple [38] scale the context-sensitive CFL-reachability analysis in a single machine and distributed environment. Chianina [39] is an out-of-core system supporting the general flow- and context-sensitive dataflow analysis.

Our work aims to tackle the poor scalability of the IFDS framework, especially when analyzing large-scale programs. Different from the existing approaches, which either rely on a heuristic-based strategy [4] or lack the dedicated design to IFDS problems [18], we propose a systematic streaming-based data-parallel DStream framework to improve the memory scalability. Moreover, thanks to the flexible control of memory consumed during computation and fine-grained parallel acceleration, DStream significantly outperforms Graspan and DiskDroid in terms of both efficiency and scalability.

VI. Conclusion

We have proposed DStream, a streaming-based data-parallel IFDS framework, for tackling the poor scalability of the IFDS framework. DStream improves the scalability from two aspects: (1) using streaming-based out-of-core computation to reduce memory footprint and I/O cost significantly, and (2) using fine-grained data-parallel computation to improve computing efficiency. Experimental results show that DStream can improve the scalability of the IFDS algorithm in large-scale programs and can outperform the state-of-the-art in terms of both efficiency and scalability.

Acknowledgment

We would like to thank Zhaogui Xu and Peng Di from Ant Group, and the anonymous reviewers for constructive comments and thoughtful feedback on earlier drafts. Our thanks also go to Lian Li from ICT CAS for providing the code of baselines. This work is supported in part by the Leading-Edge Technology Program of Jiangsu Natural Science Foundation (No. BK20202001) and the National Natural Science Foundation of China (No.62232008, No.62172200, No.62272217).

References

[1] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 1995, pp. 49–61. [Online]. Available: http://doi.org/10.1145/199448.199462

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2014, pp. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[3] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue, "Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 267–279. [Online]. Available: https://doi.org/10.1109/ASE.2019.00034

[4] H. Li, H. Meng, H. Zheng, L. Cao, J. Lu, L. Li, and L. Gao, "Scaling up the IFDS algorithm with efficient disk-assisted computing," in *Proceedings of the 19th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 236–247. [Online]. Available: https://doi.org/10.1109/CGO51591.2021.9370311

[5] S. Arzt, "Sustainable solving: Reducing the memory footprint of IFDS-based data flow analyses using intelligent garbage collection," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1098–1110. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00102

[6] Y. Li, T. Tan, Y. Zhang, and J. Xue, "Program tailoring: Slicing by sequential criteria," in *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 15:1–15:27. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2016.15

[7] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang, "PSE: Explaining program failures via postmortem static analysis," in *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2004, pp. 63–72. [Online]. Available: https://doi.org/10.1145/1029894.1029907

[8] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analyses," in *Proceedings of the 23rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2002, pp. 69–82. [Online]. Available: https://doi.org/10.1145/512529.512539

[9] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 171–180. [Online]. Available: https://doi.org/10.1145/1368088.1368112

[10] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in Android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 167–177. [Online]. Available: https://doi.org/10.1145/3238147.3238185

[11] A. Gotsman, J. Berdine, and B. Cook, "Interprocedural shape analysis with separated heap abstractions," in *Proceedings of the 13th International Symposium on Static Analysis (SAS)*. Springer, 2006, pp. 240–260. [Online]. Available: https://doi.org/10.1007/11823230_16

[12] S. Arzt and E. Bodden, "Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 288–298. [Online]. Available: http://doi.org/10.1145/2568225.2568243

[13] J. Rodriguez and O. Lhoták, "Actor-based parallel dataflow analysis," in *Proceedings of the 20th International Conference on Compiler Construction (CC)*. Springer, 2011, pp. 179–197. [Online]. Available: http://doi.org/10.1007/978-3-642-19861-8_11

[14] Q. Shi and C. Zhang, "Pipelining bottom-up data flow analysis," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 835–847. [Online]. Available: https://doi.org/10.1145/3377811.3380425

[15] D. Helm, F. Kübler, J. T. Kölzer, P. Haller, M. Eichberg, G. Salvaneschi, and M. Mezini, "A programming model for semi-implicit parallelization of static analyses," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2020, pp. 428–439. [Online]. Available: https://doi.org/10.1145/3395363.3397367

[16] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 426–436. [Online]. Available: https://doi.org/10.1109/ICSE.2015.61

[17] E. Bodden, "Inter-procedural data-flow analysis with IFDS/IDE and Soot," in *Proceedings of the 1st ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis (SOAP)*. ACM, 2012, pp. 3–8. [Online]. Available: https://doi.org/10.1145/2259051.2259052

[18] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani, "Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017, pp. 389–404. [Online]. Available: https://doi.org/10.1145/3037697.3037744

[19] Z. Zuo, K. Wang, A. Hussain, A. A. Sani, Y. Zhang, S. Lu, W. Dou, L. Wang, X. Li, C. Wang, and G. H. Xu, "Systemizing interprocedural static analysis of large-scale systems code with Graspan," *ACM Transactions on Computer Systems (TOCS)*, vol. 38, no. 1-2, pp. 4:1–4:39, 2021. [Online]. Available: https://doi.org/10.1145/3466820

[20] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," in *Program Flow Analysis: Theory and Application*. Prentice Hall, 1981, pp. 189–233.

[21] T. Reps, "Shape analysis as a generalized path problem," in *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM, 1995, pp. 1–11. [Online]. Available: https://doi.org/10.1145/215465.215466

[22] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for Java," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2005, pp. 59–76. [Online]. Available: https://doi.org/10.1145/1094811.1094817

[23] X. Zheng and R. Rugina, "Demand-driven alias analysis for C," in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2008, pp. 197–208. [Online]. Available: https://doi.org/10.1145/1328438.1328464

[24] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM (CACM)*, vol. 33, no. 8, pp. 103–111, 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[25] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction (CC)*. ACM, 2016, pp. 196–206. [Online]. Available: https://doi.org/10.1145/2892208.2892226

[26] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 2009, pp. 243–262. [Online]. Available: https://doi.org/10.1145/1640089.1640108

[27] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM, 1999, pp. 13:1–13:11. [Online]. Available: https://dl.acm.org/doi/10.5555/781995.782008

[28] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proceedings of the 12th International Conference on Compiler Construction (CC)*. Springer-Verlag, 2003, pp. 153–169. [Online]. Available: https://doi.org/10.1007/3-540-36579-6_12

[29] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Theoretical Computer Science (TCS)*, vol. 167, no. 1, pp. 131–170, 1996. [Online]. Available: http://doi.org/10.1016/0304-3975(96)00072-2

[30] T. Reps, "Program analysis via graph reachability," *Information and Software Technology (IST)*, vol. 40, no. 11, pp. 701–726, 1998. [Online]. Available: https://doi.org/10.1016/S0950-5849(98)00093-7

[31] N. A. Naeem, O. Lhoták, and J. Rodriguez, "Practical extensions to the IFDS algorithm," in *Proceedings of the 19th International Conference on Compiler Construction (CC)*. Springer, 2010, pp. 124–144. [Online]. Available: http://doi.org/10.1007/978-3-642-11970-5_8

[32] J. Lerch, J. Späth, E. Bodden, and M. Mezini, "Access-path Abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths," in *Proceedings of the 30th IEEE/ACM International Conference on*

Automated Software Engineering (ASE). IEEE, 2015, pp. 619–629. [Online]. Available: https://doi.org/10.1109/ASE.2015.9

[33] P. D. Schubert, B. Hermann, and E. Bodden, "PhASAR: An inter-procedural static analysis framework for C/C++," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2019, pp. 393–410. [Online]. Available: https://doi.org/10.1007/978-3-030-17465-1_22

[34] M. Madsen, M.-H. Yee, and O. Lhoták, "From Datalog to Flix: A declarative language for fixed points on lattices," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2016, pp. 194–208. [Online]. Available: http://doi.org/10.1145/2908080.2908096

[35] WALA: T.J. Watson Libraries for Analysis. Accessed: 2022. [Online]. Available: https://github.com/wala/WALA

[36] C. Weiss, C. Rubio-González, and B. Liblit, "Database-backed program analysis for scalable error propagation," in *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, Volume 1 (ICSE)*. IEEE, 2015, pp. 586–597. [Online]. Available: https://doi.org/10.5555/2818754.2818827

[37] Z. Zuo, R. Gu, X. Jiang, Z. Wang, Y. Huang, L. Wang, and X. Li, "BigSpa: An efficient interprocedural static analysis engine in the cloud," in *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 771–780. [Online]. Available: https://doi.org/10.1109/IPDPS.2019.00086

[38] Z. Zuo, J. Thorpe, Y. Wang, Q. Pan, S. Lu, K. Wang, G. H. Xu, L. Wang, and X. Li, "Grapple: A graph system for static finite-state property checking of large-scale systems code," in *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*. ACM, 2019, pp. 38:1–38:17. [Online]. Available: https://doi.org/10.1145/3302424.3303972

[39] Z. Zuo, Y. Zhang, Q. Pan, S. Lu, Y. Li, L. Wang, X. Li, and G. H. Xu, "Chianina: An evolving graph system for flow- and context-sensitive analyses of million lines of C code," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2021, pp. 914–929. [Online]. Available: https://doi.org/10.1145/3453483.3454085