Scaling Bottom-Up IFDS Taint Analysis with Optimized Data-Flow Encoding

Fabian Schiebel
Heinz Nixdorf Institute, Paderborn University, Germany

Eric Bodden
Department of Computer Science, Heinz Nixdorf Institute, Paderborn University, Germany
Fraunhofer IEM, Paderborn, Germany

Abstract

Static taint analysis is an important technique for bug-finding and vulnerability detection on source-code. One commonly used technique to solve inter-procedural taint analysis problems is the well-studied *Interprocedural Finite Distributive Subsets* (IFDS) algorithm. However, when performing a whole-program analysis, IFDS-based taint analyses have trouble scaling to large target programs, mainly due to its huge memory consumption. A common technique to solve this issue is to modularize the analysis by visiting the call graph *bottom-up*, performing local sub-analyses in isolation. Still, bottom-up formulations of IFDS are rare due to their own scalability issues caused by over-approximating the analysis state at the beginning of a procedure. In this work we study how one can improve scalability for bottom-up taint analysis of C and C++ programs on millions of lines of code. We present MonoIFDS, a variant of the IFDS algorithm that combines IFDS-style procedure summaries with bottom-up inter-procedural propagation. It uses a modified encoding of the data-flow state and propagation to improve the efficiency of handling large analysis states, and to enable optimizations, such as optimizing the iteration order, that were not beneficial on IFDS analyses before. In our evaluation we show, how MonoIFDS performs compared to SparseIFDS when analyzing real-world C and C++ programs. Achieving a speedup of 6.17 on average with only consuming 51% of the memory compared to SparseIFDS, MonoIFDS drastically outperforms the state-of-the-art SparseIFDS and enables developers to target the analysis of large programs on consumer hardware.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Static analysis, IFDS, C/C++

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.23

Funding This work was partially supported by the Fraunhofer institute for Mechatronic Systems Design (IEM).

1 Introduction

Static analysis is a useful technique for detecting bugs and vulnerabilities in software applications without executing them. Taint analysis is a class of static analysis commonly used to detect data leaks, double-free (CWE-415), use-after-free (CWE-416), SQL-injection (CWE-89), command-injection (CWE-77) or similar vulnerabilities. It tracks *tainted* data that originates from specified *source* statements and reports a *leak*, if *tainted* data may reach a specified *sink* location without being *sanitized* before. To not overload developers with too many false positives, a taint analysis should ideally be precise. To be precise, it must be inter-procedural, i.e., follow procedure calls to the call targets. However, existing approaches to sound and precise inter-procedural taint analysis are computationally expensive and do not scale well to large programs [2, 11, 12, 14, 15, 17].

The *interprocedural finite distributive subsets* (IFDS) framework [18, 20] is a commonly used algorithmic base to implement a taint analysis. IFDS is maximally precise: because it assumes flow functions that distribute over the merge operator, it computes solutions

© Fabian Schiebel and Eric Bodden;
licensed under Creative Commons License CC-BY 4.0

that resemble the theoretically optimal meet-over-all-paths (MOP) solution. IFDS solves a specified data-flow problem by reducing it to a graph-reachability problem. It constructs precise procedure summaries, which are also fine-grained: they are defined fact by fact, and can be reused in any context in which just a single fact matches that of a previously considered context for the same callee procedure. This way, IFDS scales relatively well in theory. Yet, experiments have shown that nonetheless real-world analyses implemented on top of IFDS require a decent amount of engineering effort to make them applicable to medium- to large-scale programs [17, 22, 32]. There are various techniques to improve the scalability of IFDS, such as SparseIFDS [12], which enables one to skip statements during analysis that are unimportant to the currently propagated data-flow fact. Despite these optimizations, our experiments failed to analyze large-scale software (>1 000 000 LOC), or required several hours of computing time, as well as dozens of Gigabytes of memory, making it impossible to target the analysis of large programs on consumer hardware. Since one cannot just add arbitrarily much memory to a consumer system, we consider the memory problem even more severe than long running times. Analysis-specific garbage collection can save memory [2, 11], however it usually requires the execution in a separate thread and may slow down execution of the baseline analysis. Also, persisting and re-using library summaries to speed up re-analysis of application code is a non-trivial problem with IFDS [24] and has non-negligible overhead on the initial run.

To address these problems, especially to lower the memory consumption, this paper presents a novel conceptual analysis framework MonoIFDS, which combines ideas of IFDS with that of bottom-up [4, 9, 10] summary-based analysis. A core idea of MonoIFDS is to split the whole-program SparseIFDS analysis into smaller chunks that can be processed in isolation: When the analysis of a chunk has finished, the solver can discard the intermediate analysis state associated with this chunk immediately without even requiring any special garbage collection.

Usually, IFDS is formulated as a top-down analysis, i.e., a procedure is analyzed *before* its callees, triggering the analysis of called procedures when the calls are encountered. This hinders divide-and-conquer, which is why MonoIFDS realizes a bottom-up analysis instead. MonoIFDS's bottom-up analysis processes strongly connected components (SCC) of the call graph (mostly single procedures) in isolation, *before* analyzing their callers. For non-recursive calls, this makes the on-demand analysis of callees obsolete and predictably limits the scope of an analyzed chunk to any such SCC. Furthermore, summaries on SCCs can be persisted individually without additional overhead.

MonoIFDS seeks to preserve from IFDS the property of conducting the analysis in a distributive way, as this guarantees both a precise analysis solution and the potential to reuse summaries on a fine-grained per-fact level. Yet, realizing an efficient bottom-up analysis comes with its own challenges, particularly when basing it on IFDS. A bottom-up analysis analyzes procedures *before* analyzing their callers. The initial analysis state at the beginning of each procedure therefore has to be approximated conservatively, which can cause the analysis to propagate more abstract state than a traditional top-down IFDS analysis. If no care is taken, this aspect could cause the bottom-up analysis to show a running time and memory consumption worse than traditional top-down IFDS. IFDS propagates each fact individually, which causes the analysis to visit the same statements multiple times, once per abstract procedure input. In bottom-up analysis the current procedure's calling context is unknown, which is why one would have to repeatedly iterate through the method for *any possible* abstract input to that method – an intractable problem with large domains that incorporate, for instance, access paths.

MonoIFDS addresses this problem by taking inspiration from monotone frameworks [13]. Specifically, it realizes an alternative design of intra-procedural data-flow propagation and encoding of flow functions, such that facts end up being propagated not individually but combined into a single data structure.

MonoIFDS thus yields the useful combination of IFDS-like summaries that can be reused on a per-fact basis, with bottom-up analysis for modularization and batch-wise data-flow propagation to efficiently handle larger analysis states.

As we show, with MonoIFDS one is also able to implement two otherwise common optimizations into IFDS, representing parts of the analysis state via bitsets, and optimizing the worklist iteration order, further improving the analysis performance in time and space.

MonoIFDS elegantly incorporates sparsification: By offloading the decision about whether a data-flow fact is needed at a certain statement onto the user-defined flow functions, and by making identity-flows implicit, we achieve a precise IFDS analysis with performance improvements comparable to SparseIFDS, without the need to at all construct sparse per-fact control-flow graphs.

These optimizations have direct impact on the constant factors of the analysis complexity, but may also influence the observed asymptotic complexity in practice.

We implement MonoIFDS in C++ based on LLVM [16] and PhASAR [23], and evaluate it against a traditional implementation of SparseIFDS as well as a direct translation from SparseIFDS to bottom-up IFDS that keeps IFDS-style data-flow propagation. The experimental results show that using MonoIFDS one can achieve an average speedup of 6.17 compared to SparseIFDS, while cutting its memory requirements in half. MonoIFDS scales better to large C and C++ programs, enabling one to analyze large target programs, such as `Z3` or `FFmpeg` on consumer hardware in reasonable time and space, which was not possible before with SparseIFDS.

In summary, this paper, as part of the proposed solution MonoIFDS, makes the following original contributions:

- It proposes an optimized data-flow encoding to build a scalable bottom-up variant of the IFDS algorithm, preserving the precision and fine-grained summary reuse of IFDS.
- It shows how to incorporate the effects of sparsification in an elegant fashion, less intrusive and more efficient than previously described.
- We evaluate our approach against the SparseIFDS baseline.

The remainder of the paper is structured as follows: Section 2 gives necessary background information about data-flow analysis and the IFDS algorithm; Section 3 presents the bottom-up IFDS formulation and details our novel MonoIFDS approach. In Section 5 we empirically evaluate the performance of MonoIFDS on real-world C and C++ applications. In Section 6 we give an overview on similar or related papers; finally, we conclude this work in Section 8.

2 Background

2.1 Basics of Data-flow Analysis

A static analysis usually works on a graph-representation of the target program. An important element is the control-flow graph (CFG), which is a directed graph-representation of the statements or basic blocks within a procedure. CFG nodes are denoted as $n \in N$. The CFG

Algorithm 1 The original IFDS algorithm from Reps et al. [20], augmented with the common on-the-fly construction from Naeem et al. [18] and sparsification adapted from He et al. [12].

```
1  while WorkList ≠ ∅ do
2      Select and remove edge ⟨s_p, d_1⟩→⟨n, d_2⟩ from WorkList;
3      switch n do
4          case n is call calling procedure q do
5              foreach d_3 ∈ passArgs(⟨n, d_2⟩) do
6                  Propagate(⟨s_q, d_3⟩→⟨s_q, d_3⟩);
7                  Incoming[⟨s_q, d_3⟩]∪ = ⟨n, d_2⟩;
8                  foreach ⟨e_q, d_4⟩ ∈ EndSummary[⟨s_q, d_3⟩] do
9                      foreach d_5 ∈ returnVal(⟨e_q, d_4⟩ , ⟨n, d_2⟩) do
10                         Insert ⟨n, d_2⟩→⟨retSite(n), d_5⟩ into SummaryEdge;
11             foreach d_3, s.t. d_3 ∈ callFlow(⟨n, d_2⟩) or ⟨n, d_2⟩→⟨retSite(n), d_3⟩ ∈ SummaryEdge do
12                 Propagate(⟨n, d_1⟩→⟨retSite(n), d_3⟩);
13         case n is exit node of p do
14             EndSummary[⟨s_p, d_1⟩]∪ = ⟨n, d_2⟩;
15             foreach ⟨c, d_4⟩ ∈ Incoming[⟨s_p, d_1⟩] do
16                 foreach d_5 ∈ returnVal(⟨n, d_2⟩ , ⟨c, d_4⟩) do
17                     if ⟨c, d_4⟩→⟨retSite(c), d_5⟩ ∉ SummaryEdge then
18                         Insert ⟨c, d_4⟩→⟨retSite(c), d_5⟩ into SummaryEdge;
19                         Set f = procOf(c);
20                         foreach d_3, s.t. ⟨s_f, d_3⟩→⟨c, d_4⟩ ∈ PathEdge do
                               Propagate(⟨s_f, d_3⟩→⟨retSite(c), d_5⟩) ;
21         otherwise do
22             foreach m, d_3, s.t. m ∈ CFG ∧ d_3 ∈ flow(⟨n, d_2⟩) do
23                 Propagate(⟨s_p, d_1⟩→⟨m, d_3⟩);
24 end
25 Procedure Propagate(⟨s_p, d_1⟩→⟨n, d_2⟩)
26     Set e = ⟨s_p, d_1⟩→⟨nextUse(n, d_2), d_2⟩;
27     if e ∉ PathEdge then
28         Insert e into PathEdge;
29         Insert e into WorkList;
```

for a procedure $p \in P$ has two special nodes, the unique start-point $s_p \in N$ and the unique exit-point $e_p \in N$. For inter-procedural analysis, we use an inter-procedural CFG (ICFG) connecting the CFGs of all (reachable) procedures with call-graph information.

Data-flow analyses work on a data-flow domain $\mathcal{D}$, which is a bounded semi-lattice with join operator $\sqcup$. Each CFG-node n has an associated *flow function* $f : \mathcal{D} \to \mathcal{D}$ that captures the effects of n on the analysis state. Each f is monotone over $\sqcup$.

2.2 The IFDS Framework

The interprocedural finite distributive subsets (IFDS) framework provides an algorithm to efficiently solve interprocedural data-flow problems in a context- and flow-sensitive way. It can be used to solve reachability-based data-flow problems with distributive flow functions that transform sets of data-flow facts $d \in D$. In the case of IFDS, the data-flow domain $\mathcal{D}$ is a subset domain 2^D and the lattice-join operator $\sqcup$ is fixed to set-union $\cup$. Also, IFDS demands flow functions to be distributive over this operator, i.e., for every flow function $f : \mathcal{D} \to \mathcal{D}$ it should hold that $\forall a, b \in \mathcal{D} : f(a) \sqcup f(b) = f(a \sqcup b)$.

The IFDS algorithm exploits the distributivity of the flow-functions by splitting them up into *micro flow functions* $\hat{f} : D \cup \{\Lambda\} \to 2^D$, s.t. $f(x) = \hat{f}(\Lambda) \cup \bigcup_{d \in x} \hat{f}(d)$ that can be represented as bi-partite graphs (cf. Definition 3.1 in [20]). Here, Λ (aka. *zero*) is a tautological data-flow fact that always holds and can be thought of as representative of the empty input set. We commonly call a micro flow function $\hat{f} : d \mapsto \{d\}$ *identity flow*, or just *identity* as it does not modify the input fact and propagates it unchanged. The IFDS algorithm builds up a so-called exploded supergraph (ESG) by incrementally composing the micro flow functions of a procedure together. The composition $\hat{f}_1 \circ \hat{f}_2$ of two (or more) micro flow functions $\hat{f}_1$ and $\hat{f}_2$ defines so-called path edges that compose the corresponding bi-partite graph-representations of $\hat{f}_1$ and $\hat{f}_2$. A path edge $\langle s_p, d_1 \rangle \to \langle n, d_2 \rangle$ is usually represented as triple $\langle d_1, n, d_2 \rangle$, since the source node s_p can be derived from n; all path edges have both source- and target in the same procedure and usually start at the beginning of a procedure. In such a path edge, we commonly call d_1 the *source fact*, d_2 the *target fact*, and n the *target node* or simply *target*. A data-flow fact $d \in D$ holds at a statement $n \in N$, if the ESG node $\langle n, d \rangle$ is reachable from some node $\langle n_0, \Lambda \rangle$, where $n_0 \in N$. Current state-of-the-art implementations of the IFDS algorithm follow this approach and propagate each path edge individually [3, 5, 23, 31].

IFDS follows the summary-based approach to achieve context-sensitive interprocedural data-flow results. It computes fine-grained per-fact summaries that can be *unconditionally* applied at subsequent calls of the same procedure. The summary of a procedure p consists of the path edges in the ESG that start at s_p and end at e_p, and therefore summarize the complete data-flow effects of p. In contrast to other approaches to context-sensitivity, e.g., k-limited call-strings [19], IFDS provides unlimited context sensitivity ($k = \infty$).

The IFDS framework was first introduced in 1995 [20] and has since then been optimized and extended in many ways, such that the plain IFDS algorithm as formulated in [20] does not represent the state of the art anymore. There are two extensions that deserve special attention: Naeem et al. [18] introduce an on-the-fly algorithm that only builds up the parts of the ESG that are needed by the analysis. He et al. [12] perform sparsification on the control-flow graph level: Since IFDS propagates each data-flow fact individually, many program statements do not interact with the propagated target fact. They introduce SparseIFDS, which skips these statements, making the analysis faster and consume less memory. Algorithm 1 provides an overview on the version of the IFDS algorithm that we build upon. Note that we exclude the final gathering of analysis results (lines 6-8 in Figure 3 of [20]). Arzt [2] argues that this additional work is not needed, especially in the scenario of taint analysis: Analysis results, i.e., detected leaks, can be reported directly when the corresponding flow function generates the right fact at a sink statement. We follow this practice.

2.3 Strongly-connected Components

Given a directed graph $G = (V, E)$. If a vertex $v \in V$ is reachable from $u \in V$ via zero or more edges from E, we denote it as $u \rightsquigarrow v$. Then, a strongly-connected component (SCC) of G is a largest subset $S \subseteq V$, s.t., $\forall u, v \in S : (u \rightsquigarrow v) \wedge (v \rightsquigarrow u)$.

The relation $(u \rightsquigarrow v) \wedge (v \rightsquigarrow u)$ is an equivalence relation [30], so there exists a partitioning on V into non-overlapping SCCs. We denote the SCC of a vertex v as $sccOf(v)$. A vertex v is part of a cycle, iff $|sccOf(v)| > 1 \vee (v, v) \in E$.

Collapsing the SCCs of a graph into single vertices, we obtain a derived graph G_{SCC}. Apart from self-loops, G_{SCC} is acyclic, allowing to compute a topological order on G_{SCC}.

3 The Bottom-Up IFDS Solver

To solve the performance issues of IFDS-based analyses, we propose to use a bottom-up strategy, simplifying the analysis with the well-known divide-and-conquer paradigm. A bottom-up analysis performs a local sub-analysis on each call-graph SCC (cf. Subsection 2.3), analyzing a procedure *before* analyzing its callers. When analyzing the callers, an already complete summary of the callee can be directly plugged-in at the respective call-sites. Except for recursive calls, it is guaranteed by construction that for a called procedure the corresponding sub-analysis has already completed, allowing the local sub-analyses to drop their intermediate state (SCC-local ESG) immediately after reaching a fixpoint. Provided that the SCCs are small, i.e., the analyzed program does not contain large mutual recursion, this has the potential of drastically reducing the memory requirements of an analysis.

In the following, we adapt the bottom-up strategy to IFDS taint analysis. Subsection 3.1 shows how we port the SparseIFDS algorithm to a bottom-up formulation. In Subsection 3.2, we present MonoIFDS, our variant of bottom-up IFDS using an improved encoding of the data-flow state and the flow functions, to overcome performance degradation by additional work that has to be done due to the bottom-up strategy. Based on that we present two optimizations in Subsection 3.3 that work well on MonoIFDS but not on regular SparseIFDS, and finally in Subsection 3.4 we examine the algorithmic complexity that may materialize in practice when using MonoIFDS instead of regular SparseIFDS.

3.1 Basic Bottom-up IFDS

In bottom-up IFDS, we first sort the call-graph SCCs in topological order, and then perform a separate forward SparseIFDS analysis on each SCC, callees before their callers. In the following, we detail on this SCC-local IFDS analysis, as we need to add some modifications for taint analysis with respect to the reporting of leaks.

Handling Recursion

One important aspect of the bottom-up IFDS algorithm is how it handles (mutual) recursion. Since we have already computed the call-graph SCCs, we can precisely detect whether a given procedure or call-site is recursive or not (see Subsection 2.3): A call-site inside procedure p that calls procedure q is recursive, iff $sccOf(p) = sccOf(q)$. Recursive call-sites are the only ones where (by construction) the callees' summaries may not be complete yet when visiting during analysis. So, we register the calling context inside the *Incoming* map, similar to usual top-down IFDS as shown in Line 6 in Algorithm 2. This ensures that the analysis can continue there with updated information, once the summaries of the callees have been updated as shown in Algorithm 3. Note that the incoming-loop in Line 8 only has an effect inside recursive procedures, as *Incoming* now is only filled from recursive call-sites. This way, we perform a SCC-local fixpoint iteration.

Leaking Taints with Approximated Sources

In contrast to normal top-down IFDS, a bottom-up analysis does not know which input data-flow facts hold: In IFDS, when a procedure is called, the data-flow facts for the passed arguments at a call-site are mapped to the respective facts for the formal parameters inside the callee (cf. *passArgs* in Algorithm 1). Due to the on-the-fly ESG construction from Naeem et al. [18], it is known by construction that each propagated data-flow fact has originally

been created from one of the input seeds for the analysis. Hence, these facts are guaranteed to hold, which enables a client taint analysis to report leaks on the fly, whenever a tainted value gets propagated into any sink. An additional reachability check is not required.

However, in the bottom-up analysis, since a procedure p is analyzed *before* analyzing its callers (except the procedures within the same SCC), it is not known in advance whether data-flow facts that are derived from formal parameters of p or from global variables may ever be generated from Λ. This has two implications: (1) All facts derived from global variables and parameters of p must be computed irrespective of whether they will later-on be determined as reachable in the overall program, and (2) taint leaks cannot be reported when facts derived from parameters or globals flow into a sink – only when leaking data that has been tainted locally. While (1) may have drastic performance impact, (2) is a critical limitation that must be solved to retain a functioning taint analysis.

To address (1), we add multiple heuristic filters to reduce the number of spurious propagated data-flow facts. On the one hand, we add type-based filtering: Only those parameters and globals are generated where their types are compatible with those created by at least one taint-source or taint-propagator. While helpful in general, his may limit the reusability of computed summaries under future changes to the application that introduce new taint sources of a different type (e.g. `int` instead of `String`). This type-based filtering is similar to type-based sparsification introduced by Karakaya et al. [14]. On the other hand, we use call-graph information to filter out globals that are never used by the current procedure or its transitive callees. Despite these filter steps, the bottom-up analysis may still need to generate more facts than a regular IFDS analysis because regular IFDS – due to its top-down nature – has precise information about which values are tainted at a procedure's call-sites. This limitation is inherent to any bottom-up approach.

To retain the ability to report taint leaks on the fly, we adapt the approach from Banerjee et al. [4] to embed potential leaks into a procedure's summary. In contrast to Banerjee et al. [4], we only store sink-effects and no source- or sanitizer effects; these are already captured in the usual IFDS summary *EndSummary*. We introduce an additional map $\mathit{LeakIf} : P \times D \to 2^{N \times D}$ storing potential leaks of the form $\langle p, d_{src} \rangle \mapsto \{\langle n_1, d_1 \rangle, \ldots, \langle n_m, d_m \rangle\}$ representing that data-flow fact d_k is leaked at statement n_k (within procedure p), if the fact d_{src} is tainted at the start of procedure p, for $k = 1, \ldots, m$. We update the *LeakIf* summary in Algorithms 2 and 3 using the PropagateLeaks procedure. At a call-site where the callee q's summary is applied, we now also apply the $\mathit{LeakIf}[q]$ summary of q by checking each mapped fact for potential leaks, as shown in Algorithm 2. Hereby, we distinguish two cases:

1. The corresponding fact from the call-site has been generated within the same procedure. This is indicated by the source fact of the current path edge being Λ. Then, the argument fact is guaranteed to hold, and the corresponding potential leaks can be reported as actual leaks.
2. The corresponding fact originates from a parameter or global. Then, it is not guaranteed that the fact ever holds, so we add all corresponding potential leaks of the callee to the potential leaks of the caller.

Note that in contrast to regular IFDS (see Line 6 in Algorithm 1), at call-sites we do *not* need to trigger the analysis of the callee-facts because either the callee is already analyzed to completion (if it is part of a different SCC), or all relevant facts have already been generated.

Similar to call-sites, we also adapt the handling of exit-sites as shown in Algorithm 3. Especially, we track the procedures where the set of potential taint-leaks has changed. This is, since – in addition to propagating a single summary edge to the callers – we also need to re-propagate the potential leaks in case they have been updated. Furthermore, it is necessary

Algorithm 2 Handling of call-sites in the bottom-up IFDS algorithm. The changed parts compared to regular top-down IFDS are highlighted in green.

```
1  while WorkList ≠ ∅ do
2      ...;
3      switch n do
4          case n is call calling procedure q do
5              foreach d3 ∈ passArgs(⟨n, d2⟩) do
6                  if sccOf(p) = sccOf(q) then
7                      Incoming[⟨sq, d3⟩] ∪= ⟨n, d2⟩;
8                  foreach ⟨eq, d4⟩ ∈ EndSummary[⟨sq, d3⟩] do
9                      foreach d5 ∈ returnVal(⟨eq, d4⟩, ⟨n, d2⟩) do
10                         Insert ⟨n, d2⟩→⟨retSite(n), d5⟩ into SummaryEdge;
11                 foreach Leaks ∈ LeakIf[⟨q, d3⟩] do
12                     PropagateLeaks(p, d1, Leaks);
13
14             foreach d3, s.t. d3 ∈ callFlow(⟨n, d2⟩) or ⟨n, d2⟩→⟨retSite(n), d3⟩ ∈ SummaryEdge do
15                 Propagate(⟨n, d1⟩→⟨retSite(n), d3⟩);
16         ...;
17 RepropagateLeaks();
18 Procedure PropagateLeaks(p, d1, Leaks)
19     if d1 = Λ then reportLeaks(Leaks);
20     else
21         LeakIf[⟨p, d1⟩] ∪= Leaks;
22         if LeakIf[⟨p, d1⟩] has changed then
23             HasNewLeaks ∪= {p};
24 Procedure RepropagateLeaks()
25     while HasNewLeaks ≠ ∅ do
26         Select and remove q from HasNewLeaks;
27         foreach d3 ↦ Leaks ∈ LeakIf[q] do
28             foreach ⟨n, d2⟩ ∈ Incoming[sq, d3] do
29                 foreach ⟨s_procOf(n), d1⟩ → ⟨n, d2⟩ ∈ PathEdge do
30                     PropagateLeaks(procOf(n), d1, Leaks);
31
```

to re-propagate pending leaks when a local fixpoint is reached (see Line 17 in Algorithm 2). Listing 1 shows an example, where RepropagateLeaks() is necessary: A valid iteration order for y is lines $\langle 2, 4, 3\rangle$, where the exit handling at line 4 has no leaks to propagate to the callers of procedure f. Line 3 then produces a potential leak $\langle foo, x\rangle \mapsto \{\langle 3, y\rangle\}$, but since the path edge $\langle 1, x\rangle \to \langle 4, y\rangle$ has already been processed, the analysis stops without invoking the exit handling again. Therefore, the leak at line 3 may never be propagated to the callers of `foo` without RepropagateLeaks(). Note that this is only problematic in recursive procedures, as only those need to schedule the analysis of their callers in the bottom-up analysis.

3.2 Improved Data-flow Encoding

As explained, despite potential advantages of bottom-up IFDS in comparison to regular SparseIFDS, the bottom-up solver is likely to propagate more data-flow facts. We next describe how we counterbalance these otherwise negative effects, presenting a variant of bottom-up IFDS that we call MonoIFDS.

```
1  foo(x) {
2    y = x;
3    if(...) sink(y);
4    return y;
5  }
```

Listing 1 Example showing the need for RepropagateLeaks()

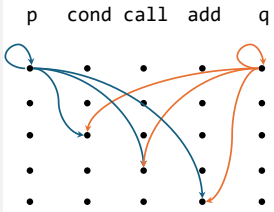
Algorithm 3 Handling of exit-sites in the bottom-up IFDS algorithm. The changed parts compared to regular top-down IFDS are highlighted in green.

```
1  switch n do
2    ...;
3    case n is exit node of p do
4      EndSummary[⟨s_p, d_1⟩]∪ = ⟨n, d_2⟩;
5      if EndSummary[⟨s_p, d_1⟩] unchanged ∧ p ∉ HasNewLeaks then
6        return;
7      HasNewLeaks = HasNewLeaks \ {p};
8      foreach ⟨c, d_4⟩ ∈ Incoming[⟨s_p, d_1⟩] do
9        foreach d_5 ∈ returnVal(⟨n, d_2⟩, ⟨c, d_4⟩) do
10         Insert ⟨c, d_4⟩→⟨retSite(c), d_5⟩ into SummaryEdge;
11         Set f = procOf(c);
12         foreach d_3, s.t. ⟨s_f, d_3⟩→⟨c, d_4⟩ ∈ PathEdge do
13           Propagate(⟨s_f, d_3⟩→⟨retSite(c), d_5⟩);
14           PropagateLeaks(f, d_3, LeakIf[⟨p, d_1⟩]);
15   ...;
```

One particular problem arises at data-flow merge points: From there on, all facts that depend on a merged-to variable have to be propagated multiple times through the rest of the procedure, once per source fact. As example, consider the LLVM IR and the corresponding ESG in Figure 1. Here, the procedure `@foo` contains a data-flow merge point at the `select` instruction `%cond`, which depends on both parameters `%p` and `%q`. Since the bottom-up analysis is conservative, it generates data-flow facts for both `%p` and `%q` at the start of `@foo` and therefore propagates facts for all values derived from `%cond`, i.e., `%cond` itself, `%call`, and `%add`, for *both* source facts `%p` and `%q`. This redundant computation may be wasteful, especially if not both `%p` and `%q` are actually tainted, leading to unnecessary performance overhead. Even with sparsification, the facts derived from `%cond` must be propagated twice at least to the instructions at which they are used. Note that this problem may also materialize in regular top-down SparseIFDS, but since a SparseIFDS taint analysis only generates facts for the parameters (and globals) that *are* tainted, this is much less relevant there.

Our goal is now to design a variant of the IFDS algorithm that combines the memory advantages of bottom-up IFDS with runtime benefits of SparseIFDS. We have already seen that the point-wise propagation of the IFDS algorithm can introduce wasteful computations. In addition, point-wise propagation prevents certain optimizations on the data-structures that hold the analysis state (*PathEdge* and *EndSummary*): When propagating each path edge individually, the only relevant operation on *PathEdge* is single-element insertion. However, for advanced data structures, e.g., bitmaps, or immutable sets, bulk-insertion and set-union can often be implemented much more efficiently than single-element insertion in a loop. For instance, computing the set-union of two bitsets can be accomplished with a bitwise-or

```
define ptr @bar(ptr %r) {
  ret ptr %r
}
define ptr @foo(ptr %p, ptr %q) {
  %cmp = icmp ne ptr %q, null
  %cond = select i1 %cmp, ptr %q, ptr %p
  %call = call ptr @bar(ptr %cond)
  %add = getelementptr i32, ptr %call, i64 1
  ret ptr %add.ptr
}
```



Initial: $p \mapsto \{p\}, q \mapsto \{q\}$

Read	Write
-	-
p, q	$cond \mapsto \{p, q\}$
$cond$	$call \mapsto \{p, q\}$
$call$	$add \mapsto \{p, q\}$

Figure 1 Exemplary LLVM IR (left), the ESG of the local SparseIFDS taint analysis on `@foo` (middle), and the corresponding behavior of the MonoIFDS flow functions (right).

operation that can be well SIMD vectorized, potentially achieving high speedups compared to inserting each element individually in a loop. For highly-efficient data-flow analysis, we aim to exploit such optimizations.

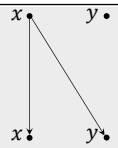
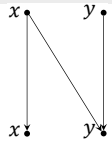
To address this, we transform the way how intra-procedural data-flows are represented and propagated, similar to monotone frameworks [13]. Specifically, we move away from the "micro" flow functions from the original IFDS algorithm (see Subsection 2.2) that are invoked for every fact at each statement and go back to flow functions that are invoked *once* per incoming data-flow *state* (usually an environment/mapping) at a statement. Importantly, we retain IFDS's procedure summaries and unlimited context-sensitivity, and full precision because we continue to demand that flow functions distribute over the join operator.

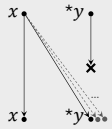
To support efficient summary computation, we modify the data-flow domain that defines the type of domain on which the flow functions operate, and influences the representation of the intermediate analysis states (*PathEdges* and *EndSummary*). We define $\mathcal{D} \subseteq D \to 2^{D \cup \{\Lambda\}}$ to store both source- and target facts (instead of only target facts) within a data-flow environment $env \in \mathcal{D}$, such that for each true fact $d_{tgt} \in D$ the analysis knows the source facts $d_{src} \in D \cup \{\Lambda\}$, from which they were generated. For example, we can encode a set of path edges $\{\langle p, n, p\rangle, \langle p, n, q\rangle, \langle \Lambda, n, q\rangle\}$ as data-flow environment $\{p \mapsto \{p\}, q \mapsto \{p, \Lambda\}\}$ at statement n. The domain $\mathcal{D}$ defines a bounded join-semi-lattice when combined with the join operation $\sqcup$ defined as map union $env_1 \sqcup env_2 := \{x \mapsto env_1[x] \cup env_2[x] \mid x \in \mathrm{dom}(env_1) \cup \mathrm{dom}(env_2)\}$. The domain is bounded because the number of different data-flow facts $|D|$ is bounded.

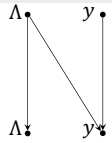
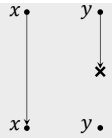
Implicit sparsification. As an optimization, we use in-place updates to the data-flow environments; the analysis' fixpoint iteration then introduces safety copies of the data-flow environment where necessary. Note that the introduced safety copies of the incoming data-flow environment are likely quicker to produce than propagating each fact individually as IFDS does, since bulk-copies can profit from cache-locality and potential SIMD vectorization, depending on the internal data-structure used for data-flow environments.

This results in *implicit sparsification*: A flow function exactly knows which facts need to be transformed and leaves all others unchanged, i.e., the identity-flow is implicit. Interestingly, one thus achieves a performance improvement comparable to the fact-specific sparsification in SparseIFDS [12], without the overhead of generating fact-specific sparse control-flow graphs. MonoIFDS's compact representation propagates a single data-flow environment containing all known facts per statement, whereas SparseIFDS propagates each data-flow

Table 1 Common intra-procedural flow functions represented in the form of a bi-partite graph, encoded as regular IFDS micro flow function, and within our modified data-flow domain. Here $pt(y)$ queries the (may-)points-to set of y and $mpt(y)$ queries the must-points-to set of y.

Stmt	Graph	Micro-FF	Mono-FF
strong assign `y=x`		$\hat{f}(d) = \begin{cases} \{x, y\} & \textbf{if } d = x \\ \emptyset & \textbf{if } d = y \\ \{d\} & \textbf{else} \end{cases}$	`f(env):` `  env[y] = env[x]`
weak assign `y=x`		$\hat{f}(d) = \begin{cases} \{x, y\} & \textbf{if } d = x \\ \{d\} & \textbf{else} \end{cases}$	`f(env):` `  env[y] ∪= env[x]`

store `*y=x`		$\hat{f}(d) = \begin{cases} \{x\} \cup pt(y) & \textbf{if } d = x \\ \emptyset & \textbf{if } y \in mpt(d) \\ \{d\} & \textbf{else} \end{cases}$	`f(env):`
			`  for d ∈ mpt(y):`
			`  env[d] = env[x]`
			`  for d ∈ pt(y)\mpt(y):`
			`  env[d] ∪= env[x]`

gen `y`		$\hat{f}(d) = \begin{cases} \{y, \Lambda\} & \textbf{if } d = \Lambda \\ \{d\} & \textbf{else} \end{cases}$	`f(env):` `  env[y] ∪= {Λ}`
kill `y`		$\hat{f}(d) = \begin{cases} \emptyset & \textbf{if } d = y \\ \{d\} & \textbf{else} \end{cases}$	`f(env):` `  env[y] = ∅`

fact separately only to statements where they are used. Both techniques essentially reduce the number of propagations (i.e, number of facts × number of statements) compared to the original IFDS. As an example, consider Figure 1 again. The right-hand side of the figure shows how the MonoIFDS flow functions on the exemplary LLVM IR snippet behave. At each statement, the corresponding flow function receives a reference to the whole incoming state *env*. The table on the right shows which data-points of *env* are queried and modified. Consider the `getelementptr` instruction that produces the value `add`: The incoming data-flow state is $env = \{p \mapsto \{p\}, q \mapsto \{q\}, cond \mapsto \{p, q\}, call \mapsto \{p, q\}\}$. Still only $env[call]$ needs to be queried, and only the new data-point $add \mapsto \{p, q\}$ is inserted, completely ignoring the other entries for p, q, and $cond$. Table 1 shows an example of how common intra-procedural flow functions can be encoded in as IFDS micro flow functions and this data-flow environment. Note that the IFDS micro flow functions are separately invoked for *every* incoming fact at a statement, even if it is the identity flow, while our transformed flow functions, called only once for the entire environment at each statement, only access the facts `x` and `y` that are needed at the current statement, and ignore all others. In particular, imagine a method in which IFDS would need to propagate plenty of facts *other than* `x` and `y`, using fact-specific identity functions for those others facts. Due to the chosen design, these other facts are then propagated implicitly, and thus with zero additional overhead. The modified MonoIFDS flow functions remain distributive over the join operator. We give an algorithmic overview of MonoIFDS in Algorithm 4.

Algorithm 4 Overview on the MonoIFDS algorithm, where intra-procedural propagations are based on the monotone framework.

Input : $CurrSCC$ – The SCC to be analyzed
Output : $EndSummary$ – Summaries of all procedures in $CurrSCC$

```
1  WorkList = {s_p | p ∈ CurrSCC};
2  foreach p ∈ CurrSCC, b ∈ blocksOf(p) do
3      AnalysisState[startOf(b)] =
         {d ↦ {d} | d ∈ seedsOf(p)} if b = startBlockOf(p) else ∅;
4  while WorkList ≠ ∅ do
5      Select and remove a node n from WorkList;
6      env = clone AnalysisState[n] ;
7      AnalyzeBlock(n, env);
8      foreach blockOf(n) → b_succ ∈ CFG do
9          AnalysisState[startOf(b_succ)] ⊔= env;
10         if AnalysisState[startOf(b_succ)] has changed then
11             Insert startOf(b_succ) into WorkList;
12 RepropagateLeaks();

13 Procedure AnalyzeBlock(n_start, env)
14     foreach n ∈ blockOf(n_start) not before n_start in CFG order do
15         switch n do
16             case n is call calling procedure q do
17                 AnalyzeCall(n, p, q, env);
18             case n is exit node of p do
19                 AnalyzeExit(n, p, env);
20             otherwise do flow(n, env);
21 Procedure AnalyzeCall(n, p, q, env)
22     if sccOf(p) = sccOf(q) then
23         Incoming[q] ∪= {n};
24         AnalysisState[n] ⊔= env;
25     NewEnv = ∅;
26     foreach d_sum ↦ Src ∈ EndSummary[q] do
27         SrcFacts = ⋃_{d3∈Src} ⋃_{d2∈passArgs^-1(e_q,d3,n)} env[d2];
28         if SrcFacts ≠ ∅ then
29             foreach d ∈ returnVal(e_q, d_sum, n) do
30                 NewEnv[d] ∪= SrcFacts;
31             foreach d1 ∈ SrcFacts do
32                 PropagateLeaks(p, d1, LeakIf[⟨q, d_sum⟩]);
33     CallFacts = ⋃_{d∈dom(env)} callFlow(n, d);
34     env = NewEnv ⊔ (env ↾_CallFacts);
35 Procedure AnalyzeExit(n, p, env)
36     EndSummary[p] ⊔= env;
37     if EndSummary[p] has changed ∨ p ∈ HasNewLeaks then
38         HasNewLeaks = HasNewLeaks \ {p};
39         WorkList ∪= Incoming[p];
```

Procedure-local Data-flow

In contrast to SparseIFDS or bottom-up IFDS, the worklist does not contain path edges, but program statements. The *AnalysisState* contains a copy of the data-flow environment for each statement where this information is needed (see below). For simplicity, we assume

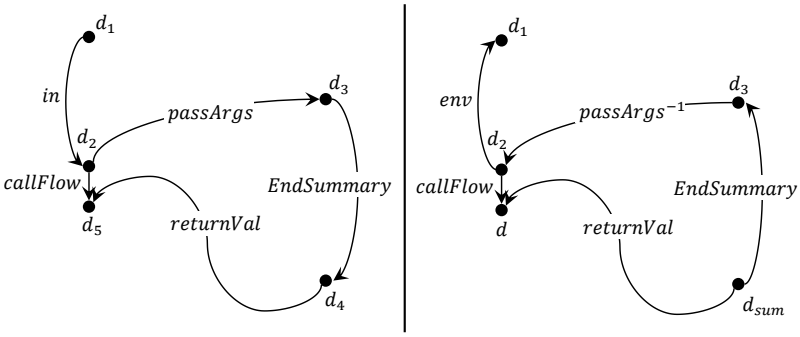


Figure 2 Schematic representation of the differences in summary application in regular IFDS (left) and MonoIFDS (right). The names corresponds to those used in the respective algorithms, Algorithm 1 for IFDS and Algorithm 4 for MonoIFDS. *in* denotes the current incoming path edge fetched from the worklist.

the CFG to be defined on basic-block level, i.e., instruction sequences without control-flow branching or merge points in the middle. This simplifies the selection of statements where a copy on the data-flow environment needs to be stored within the *AnalysisState* map. We observe that within a basic block no copies are necessary as the original state is never needed anymore after a flow function invocation. However, there are two exceptions: (1) At the start of each basic block one must create a new data-flow environment which all incoming states are merged into using map union; (2) one must also clone the data-flow environment for each potentially recursive call-site (see Line 24), i.e., each call-site that may call into the same SCC. This is to allow the return-flow-handling to re-apply an updated procedure summary.

The pseudo-code in Algorithm 4 reflects this behavior: When starting to analyze a basic block, it first fetches the currently most recent incoming data-flow environment and introduces a safety copy *env* of it (see Line 6). Then, in AnalyzeBlock the statements of the analyzed block are analyzed in order, mutating the environment *env* in place. When leaving the basic block in Line 8 and following, it updates the incoming states of each successor block and schedules (re-)analysis of each of the successors, where the incoming state has changed. As one needs to re-propagate updated summary information at potentially recursive call-sites, it is necessary to allow starting AnalyzeBlock in the middle of a basic block (see Line 14).

Since the flow functions are monotone over the join operator, the analysis will eventually reach a fixpoint and terminate. Because the flow functions are not just monotone but even distributive, one moreover obtains the precise MOP solution.

Interprocedural Data-flow

Due to our above observation that the flow functions are independent of source facts, we can modify the signature of the user-provided function *returnVal*, removing the source fact from its parameter-list. Otherwise, the inter-procedural data-flows (at call- and exit sites) are similar to the regular bottom-up flow functions, as depicted in Algorithms 2 and 3. One notable difference in AnalyzeCall is in summary application. Since our data-flow environment effectively reverses the path-edges direction, we need small modifications in call-handling. In regular IFDS, summary-application works as follows (depicted on the left side of Figure 2): First, the target facts d_2 at the call-site are mapped into the callee via *passArgs*(), these are then the source facts d_3 inside the callee that are used to lookup the summary facts d_4 via *EndSummary*; finally, the summary facts are mapped back at the call-site via *returnVal*().

In our MonoIFDS-representation of *PathEdges* and therefore also *EndSummary* the edges are inverted, so efficient lookup of summary facts in *EndSummary* is not possible. Instead (depicted on the right side of Figure 2), we start with the summary facts d_4, lookup the corresponding source facts d_3 in the callee via *EndSummary*, and map both the summary facts and their source facts back to the call-site via *returnVal*() and $passArgs^{-1}$(), respectively (Lines 29 and 27 in Algorithm 4). Here, $passArgs^{-1}$ is the inverse of the *passArgs* function. We use $passArgs^{-1}$ to filter for incoming facts that may actually hold at the call-site, by checking whether a summary fact d_{sum} has at least one caller-source fact d_1 which it may be generated from (see Line 28). Line 34 uses *callFlow* to intra-procedurally propagate facts that are unrelated to the call as identity. Hence, users of MonoIFDS need to provide *flow*, *callFlow*, *returnVal*, and $passArgs^{-1}$, where the latter can oftentimes reuse functionality from *returnVal* (e.g., when dealing with output- or in-out parameters).

3.3 Optimized Fixpoint Computation

When formulating the analysis within MonoIFDS, we can apply optimization techniques commonly referred to in the literature for monotone frameworks and abstract interpretation to speed up the fixpoint computation.

One notable optimization is to use bitsets to represent sets of source facts in the data-flow environment. From the algorithm we know that the most common operations on the data-flow environment are cloning and merging which are both very efficient on bitsets, allowing to merge sets with up to 256 or even 512 elements in a single bitwise-or operation (depending on the SIMD-register width of the underlying hardware). Similarly, using bitsets can lead to memory savings as well, since we can store up to 8 source-facts in a single byte, whereas with hashsets we would need at least one byte per source-fact (provided we always have at most 255 source-facts per procedure, which is unrealistic in real-world scenarios). In our evaluation we empirically show the performance impact of using a bitset instead of a hashset.

Note that switching from bitset to hashset helps to estimate the expected slowdown when generalizing MonoIFDS to IDE problems [21]: In IDE, path edges are annotated with edge-functions that allow encoding an additional value-computation problem. Therefore an obvious extension would change the type of data-flow environments from $\mathcal{D} : D \to 2^{D \cup \{\Lambda\}}$ to $\mathcal{D}_{IDE} : D \to (D \cup \{\Lambda\} \to J)$, where J is the domain of edge-functions. Due to the mapping to edge-functions, one could then no longer use a bitset in this setting.

Also note that applying the bitset optimization to SparseIFDS would not be beneficial: on the one hand the top-down propagation makes it infeasible to invert the path edges' direction (then the necessary access-patterns on the *PathEdges* data-structure would become inefficient [22]); furthermore, due to the fact-wise propagation the relevant set-operation would only be single-element insertion, completely defeating the main motivation of using bitsets in the first place: Speeding-up set-union with bitwise-or. On the other hand, storing target facts instead of source facts in bitsets is more challenging as the complete set of target facts only gets known *during* the analysis, i.e., it is additional overhead to map the target facts for a procedure to a contiguous range of integers, and their domain is typically larger (source facts are restricted to parameters and used global variables, whereas target facts can also be derived from local variables).

Another important optimization is the iteration order over the worklist. In regular (sparse) IFDS each fact is propagated individually, leading to a binary lattice $\begin{smallmatrix}\top\\\bot\end{smallmatrix}$ per data-flow fact where a fact either holds ($\top$) or does not hold ($\bot$); likewise, a path edge is either present, or not. Hence the propagation order does not influence the number of iterations needed to converge to a fixpoint. In contrast, for MonoIFDS the iteration order may actually

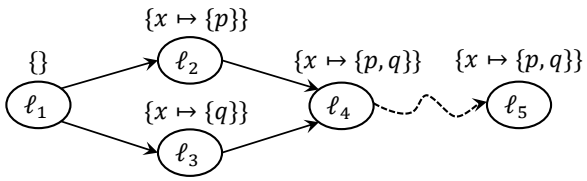


Figure 3 Exemplary control-flow graph with annotated taints, showing the relevance of optimizing the worklist iteration order.

matter, as due to the map-like data-flow environment the lattice-height now depends on the number of possible insertions into the map. Depending on the iteration order, the propagated environment may be more or less "complete" when continuing propagation after a merge point, influencing how often a statement may need to be re-analyzed with an updated environment. For example, consider Figure 3. It shows an exemplary CFG for a taint analysis, where in the branch ℓ_2 the taint x is generated from p, and in branch ℓ_3 x is generated from q. There are multiple possible iteration traces: $\langle \ell_1, \ell_2, \ell_4, \dots \ell_5, \ell_3, \ell_4, \dots, \ell_5 \rangle$, or $\langle \ell_1, \ell_2, \ell_3, \ell_4, \dots \ell_5 \rangle$. Whereas the second trace is optimal as it only visits each block once, the first one needs to repeat the analysis of ℓ_4 to ℓ_5 with updated information: the first time of visiting ℓ_4 to ℓ_5 only the taint $x \mapsto \{p\}$ is known, then after visiting ℓ_3 the additional taint $x \mapsto \{q\}$ is known and needs to be propagated to all successors. The second trace avoids the repeated analysis of ℓ_4 to ℓ_5 by delaying the analysis of the merge-point ℓ_4 until all predecessors have been analyzed to completion. With (sparse) IFDS instead, each path edge is propagated individually, so we then must analyze ℓ_4 to ℓ_5 twice in any case: once with p as source fact, and once with q.

The relevance of worklist iteration order is also shown in the literature, e.g., by Cooper et al. [8] and Bourdoncle [6], who optimize the iteration strategy to make the analysis converge faster towards a fixpoint. We adopt their ideas by computing an approximation of a topological order over a procedure's local control-flow graph. For that, we compute a reverse post-order (RPO) over the local CFGs and use a priority worklist based on this information. However, in case of (mutual) recursion, there is no statically pre-defined topological order, so we do not expect the RPO to help the analysis' convergence. Hence, we offer a variant that only uses RPO in singleton call-graph SCCs. In our evaluation, we empirically determine, how the RPO strategy performs on our real-world benchmarks.

3.4 Algorithmic Complexity

For the original IFDS algorithm the asymptotic runtime complexity is in $\mathcal{O}(|N| \cdot |D|^3)$ and its space complexity is in $\mathcal{O}(|N| \cdot |D|^2)$, driven by the maximum size of the constructed exploded supergraph [20]. Here, N is the set of ICFG nodes (i.e., statements in the program), and D is the set of all data-flow facts. Since the bottom-up variants are still implementing IFDS and use the exact same helper-analysis information (call graph, alias-information), these complexity classes build the basis for their complexities as well. Especially, the bottom-up analyses do not explore procedures that cannot be explored by the original IFDS: SCCs are only computed for the procedures that are transitively reachable from the analysis' entry-points, based on the used call graph. Furthermore, the bottom-up analyses may not generate data-flow facts that cannot be generated by the original IFDS: The only additional generated facts are the parameters and globals at the start of a procedure. They all may also be generated by the original IFDS through the *passArgs* flow function; although at runtime the number of facts generated by original IFDS may be less, this cannot influence the worst-case complexity.

The complexity of the SCC computation is bounded by the size of the call graph, which is bounded by the number of call-edges, which is in $\mathcal{O}(|N|^2)$, leading to an increased runtime complexity of $\mathcal{O}(|N|^2 + |N| \cdot |D|^3)$ for the bottom-up analyses.

However, in the MonoIFDS variant of the algorithm, the complexity may be lower in practice: Firstly, for computing the SCCs, since most call-sites are *direct* calls, especially in C and C++, we expect a runtime in $\mathcal{O}(|N|)$ (this also applies to the regular bottom-up IFDS variant). Secondly, the number of source facts per procedure is known in advance; it is the size of the set $seedsOf(p)$ for a procedure p. Since we store the source facts as sets per target fact in MonoIFDS, we can compactly represent a set of source facts using a bitset (see Subsection 3.3). Provided that the number of source facts per procedure is usually small in practice (many procedures have only a small number of parameters and do not use many global variables), we argue that both the set-operations (copy and union), and the space requirements for these sets are effectively constant.

Based on this argumentation, the complexity of the MonoIFDS algorithm *observed in practice* may be in $\mathcal{O}(|N| \cdot |D|^2)$ in time and $\mathcal{O}(|N| \cdot |D|)$ in space. We call this the *effective complexity*. In our evaluation (see Section 5) we empirically benchmark, whether these assumptions are realistic when analyzing real-world C and C++ programs.

4 Implementation

We have implemented the state-of-the-art SparseIFDS, as well as bottom-up IFDS and MonoIFDS with ≈1.5k lines of C++ using the PhASAR framework to analyze LLVM IR. We use a VTA [29] call graph that is based on PhASAR's call-graph construction and use Tarjan's algorithm [30] to compute the strongly-connected components on the call graph.

In both SparseIFDS and bottom-up IFDS we implemented the on-the-fly ESG construction by Naeem et al. [18] and sparsification adapted from He et al. [12]. To allow for fair comparisons, we share as much of the algorithmic parts as possible between the IFDS implementations. Especially the flow functions and the Propagate procedure reuse the same implementation for both SparseIFDS and the bottom-up IFDS; MonoIFDS requires some special handling for the flow functions to implement in-place updates on the data-flow environment; still the underlying flow-function logic is the same. For MonoIFDS, we use an `llvm::SmallBitVector` to store source facts in a data-flow environment that is represented with an open-addressing hash map. With a small-bit-vector, we account for our expectation that in most cases the number of source facts per procedure is small.

To be able to evaluate the solvers, we implemented a client taint analysis adapted from PhASAR's `IFDSTaintAnalysis`, configured to detect double-free errors. Hence, we configure the functions `free` and `operator delete` as both source and sink; there are no sanitizers. The taint analysis is equipped with equality-based Steensgaard-style alias information provided by LLVM [33, 34]. (We tried to use an Andersen-style [1] alias-analysis that also comes with LLVM [34], but especially for the larger target programs, we found this analysis to be not scalable enough, so we use the Steensgaard-style alias analysis instead.)

We plan to make our implementation available as open-source within the PhASAR framework in GitHub.

5 Evaluation

To test our bottom-up IFDS formulation, we perform an empirical evaluation on real-world C and C++ programs. Next, we formulate the research questions the evaluation addresses.

5.1 Research Questions

Performance of the Bottom-up Solver. The primary motivation of defining the bottom-up IFDS formulation is the performance and scalability of the IFDS algorithm. Hence, we ask:

RQ_1 *When for analyzing real-world C and C++ programs, how do the running time, memory consumption, and number of data-flow propagations of an IFDS-based taint analysis change, when switching the underlying algorithm from the baseline* SparseIFDS *to bottom-up IFDS and* MonoIFDS*?*

Complexity Reduction. In Subsection 3.4, we argue that the *effective complexity* of the MonoIFDS variant may be smaller than the known worst-case complexity of IFDS, iff the number of source facts per procedure is small. We ask:

RQ_2 *How frequently is the number of source facts per procedure small enough that the effective complexity of* MonoIFDS *drops to* $\mathcal{O}(N \cdot D^2)$ *in time and* $\mathcal{O}(N \cdot D)$ *in space?*

Impact of Optimizations on MonoIFDS. In Subsection 3.3, we propose some optimizations on the IFDS algorithm via MonoIFDS that are well-known for analyses in monotone frameworks, in the expectation of performance benefits compared to a base-implementation. Hence, we ask:

RQ_3 *What is the impact of the proposed optimizations (using bitsets for source facts and optimizing the worklist-iteration order), when using* MonoIFDS *to analyze real-world C and C++ programs?*

5.2 Experimental Setup

To allow for easy reproducibility, in what follows, we detail how we set up our experiments.

5.2.1 Benchmark Programs

We performed our experiments on the 29 real-world C and C++ programs listed in Table 2. We included the commonly analyzed programs from the GNU Core Utilities, together with programs analyzed in other static analysis papers [22, 23] to ensure a wide coverage of different domains and sizes. Especially, we included programs with $>$ 1Mio LLVM instructions, to show the scalability of the tested analyses.

5.2.2 Measurement Setup

To make the 29 target programs ready for analysis, we compiled them to LLVM-15 IR using WLLVM[1]. We used static linking whenever possible to create a single LLVM module that contains the whole program. To reduce measurement bias, we ran each experiment (IFDS-solver $\times$ target program) 10 times and report average values and standard deviation.

We used the GNU `time` core-utility to measure the total runtime and peak memory usage for all experiments. To compute more fine-granular timings, we instrumented our analysis driver tool that invokes the analyses. Our experiments are constrained by a time-limit of 60 minutes and a memory-limit of 32GiB. For each experiment that finished within the

[1] WLLVM: `https://github.com/travitch/whole-program-llvm`

Table 2 Real-world C and C++ programs targeted by our evaluation together with their domain, and the numbers of LLVM IR instructions, procedures, global variables, and call-sites. These metrics may influence the performance of the tested analyses.

	Revision	Domain	LLVM Insts	Procs	Globals	Calls
whoami	`6d683a1d`	Coreutils	5 157	174	17	923
kill	`6d683a1d`	Coreutils	5 941	188	18	1 073
cat	`6d683a1d`	Coreutils	6 683	209	24	1 188
fold	`6d683a1d`	Coreutils	6 740	204	23	1 166
uniq	`6d683a1d`	Coreutils	7 649	219	29	1 381
join	`6d683a1d`	Coreutils	9 845	245	43	1 736
wc	`6d683a1d`	Coreutils	10 347	269	42	1 844
dd	`6d683a1d`	Coreutils	19 646	313	72	3 043
cp	`6d683a1d`	Coreutils	23 619	494	38	4 524
du	`6d683a1d`	Coreutils	25 753	471	45	4 235
ls	`6d683a1d`	Coreutils	30 513	634	112	5 323
mjs	`2.20.0`	Interpreter	38 268	470	14	6 923
libpcap	`1.11.0`	Network	55 092	647	27	8 408
libpng	`a37d4836`	File format	67 601	560	1	8 142
lrzip	`ed51e14a`	Compression	76 652	691	36	9 536
readelf	`2.26`	Binutils	102 537	592	217	14 974
bison	`849ba01b`	Parser	118 632	1 260	302	19 679
lepton	`f34c7f42`	Compression	233 459	4 608	203	49 312
cxxfilt	`2.26`	Binutils	335 889	2 111	121	39 787
libgit2	`1.9.1`	Version control	453 633	5 469	151	71 782
libpoppler	`24.07.0`	Rendering	565 293	16 884	102	135 642
asterisk	`23.1.0`	Signal Processing	628 477	8 526	1 021	125 544
redis-server	`8.0.3`	Database	665 243	9 935	1 777	125 599
downward	`24.06.1`	Planning	987 053	44 376	183	297 850
opencv	`6b53fe8f`	Comp. Vision	1 171 838	20 508	1 757	280 047
openssl	`3.6.0`	Crypto	1 285 462	21 350	711	249 783
phasar-cli	`v2510`	Analysis	1 488 532	63 141	234	428 984
ffmpeg	`n4.4.6`	Multimedia	4 653 720	35 269	2 498	632 078
z3	`4.15.4`	Solver	5 188 001	122 519	598	1 511 490

given limits, we performed an additional, not measured, analysis run to validate that both respective configurations/solvers produced the same observable output (in this case: the same taint-leaks). We conducted our evaluation on a developer machine running an AMD Ryzen 9 7900X processor and assigned 32GiB GiB of RAM. To reduce external factors that may influence our measurements, we made sure that our system has exclusive access to the underlying hardware and that no other unneeded processes ran in parallel, to prevent scheduling issues.

In the following, we describe how we design our experiments to answer the research questions defined in Subsection 5.1.

Operationalization RQ_1**.** To evaluate, how the performance differs between the SPARSE-IFDS solver and our bottom-up IFDS solver, we ran all three IFDS variants on the 29 target programs. The MONOIFDS solver was benchmarked in its default configuration, using bitsets to represent source-fact sets and utilizing a de-duplicated FIFO-queue as worklist. We recorded the running time, the memory usage, and the number of propagations. For measuring the number of data-flow propagations, we instrumented the IFDS implementations with a simple counter within the Propagate procedure of the algorithm.

Operationalization RQ_2**.** To measure how often the effective complexity of MONOIFDS reduces, one first needs to define what number of source facts of a procedure is considered *sufficiently small*. In Section 4 we state that we use an `llvm::SmallBitVector` to store

source facts. This implementation can store up to 57 bits inline (on a 64-bit system) without allocating dynamic memory on the heap. So, one suitable threshold would be the ≤ 57 *source facts per procedure*. However, when analyzing real-world C and C++ applications, we consider 57 a too small number: Especially in the presence of global variables one likely needs to generate more than 57 source facts per procedure. When exceeding 57 bits, the `llvm::SmallBitVector` allocates an `llvm::BitVector`, which itself can store up to 384 bits inline (both numbers hold for 64Bit systems). Hence we set the threshold to measure to ≤ 384 *source facts per procedure*.

Based on this, we run all three IFDS variants on the benchmark programs, and add a post-processing step to measure the above defined counters. We consider our complexity assumption to hold if there are *no* procedures with more than 384 source facts.

Operationalization RQ_3. To evaluate the impact of the proposed optimizations to MonoIFDS, we run different variants of the MonoIFDS solver on the same benchmark programs. The MonoIFDS variants modify two parameters: (1) The set type for storing source facts in the data-flow environment: either a hashset or a bitset; and (2) the worklist type: either a FIFO-queue, or a priority-queue based on the RPO on the local CFG, or a hybrid using RPO for call-graph SCCs that are not mutually recursive and a FIFO-queue otherwise. The latter strategy assumes that for mutual recursion the overhead of computing a fixed ordering and using a priority-queue as worklist exceeds potential benefits. All three worklists use de-duplication.

As baseline, we use a bitset for storing source facts and a FIFO-queue as worklist. We measure running time, memory consumption, and the number of worklist iterations, and compare these with the baseline; for all three metrics, lower is better.

5.3 Evaluation Results

When running the taint analysis on the real-world programs from Table 2, the solvers terminated within the given time- and memory limit for all target programs, except for `Asterisk`, `OpenSSL`, `FFmpeg`, and `Z3`. Except for bottom-up IFDS on `Asterisk`, the solvers ran out of memory there. We ignore these targets, when computing speedups and relative memory consumption. In the following, we detail the results for our research questions.

5.3.1 Results for RQ_1

Figure 4 (left) shows the running times of the three solvers averaged over the 10 repetitions. On average, the bottom-up-IFDS achieves a speedup of 0.99 ± 0.25 (max 1.67 for `cat`) compared to regular SparseIFDS; so, on average it performs similar to the regular top-down SparseIFDS with some cases where it is slower (e.g,. for `libgit2`) and other cases where it is faster (e.g., for `bison`). In contrast, MonoIFDS achieves a speedup factor of 3.23 ± 2.12 (max 12.9 for `redis-server`) compared to SparseIFDS. Except for one target, `lrzip`, MonoIFDS was always faster than both SparseIFDS and bottom-up IFDS. This is likely because the regular SparseIFDS is very sparse for `lrzip` propagating only 30 130 path edges, whereas the bottom-up analyses compute 2 894 164 path edges. For all other targets, the ratio *path edges/instructions* is very similar for both bottom-up and SparseIFDS with bottom-up IFDS propagating slightly more path edges than SparseIFDS.

Figure 4 (right) shows the average memory consumption of the three IFDS solvers. On average, bottom-up IFDS consumes $77.3\% \pm 24.0\%$ (min 32.6% for `OpenCV`) of the memory that SparseIFDS consumes, whereas MonoIFDS consumes on average $51.3\% \pm 27.4\%$

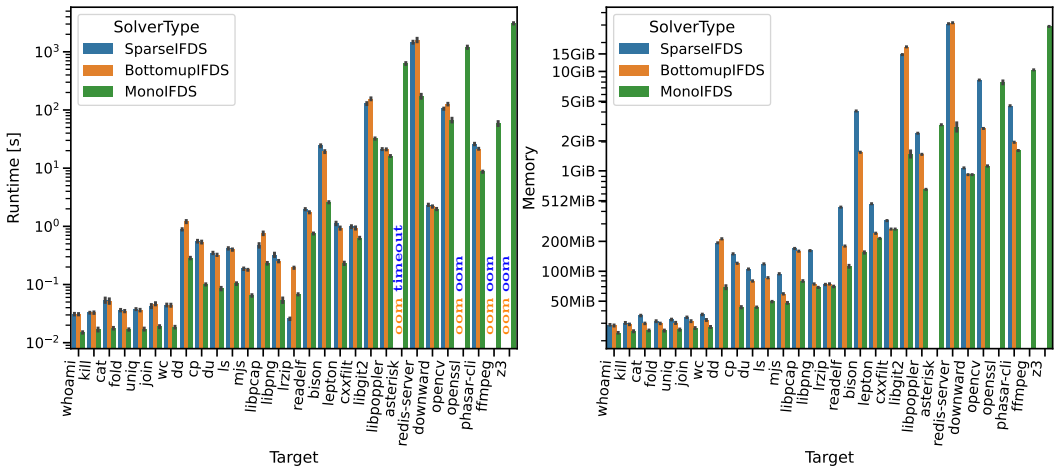


Figure 4 Running time (left) and memory consumption (right) of SparseIFDS, bottom-up IFDS and MonoIFDS. Both graphs show their metrics on a log-scale. Lower is better. The black bars denote the standard deviation and the target programs are sorted by number of LLVM instructions in ascending order. The labels *oom* and *timeout* denote that the respective analysis solver exceeded the given memory- or time limit.

(min 3.6% for `bison`) of the memory usage of SparseIFDS. In contrast to MonoIFDS, the relative performance of bottom-up IFDS is not stable: For some programs it performs better than SparseIFDS, for others it performs worse. Figure 5 shows the number of path edges propagated by the IFDS solver variants. Note that we did not measure the number of path edges for MonoIFDS as it propagates whole data-flow environments and thus potentially many path edges at a time. For MonoIFDS we measured the number of worklist iterations instead that are relevant for RQ_3. For many target programs SparseIFDS and the bottom-up variants propagate a similar number of path edges, indicating negligible overhead induced by the over-approximation of source facts at the beginning of the procedures. For some targets, however, the bottom-up analyses must propagate significantly more path edges, e.g., for `lrzip` or `libgit2`. For these targets Figure 4 also shows better performance of SparseIFDS compared to bottom-up IFDS, and for `lrzip` even to MonoIFDS.

The basic bottom-up IFDS encoding performs worse than SparseIFDS in both runtime and memory; only in a few cases bottom-up IFDS can hold to the expectation of lower memory consumption. MonoIFDS successfully solves these issues, achieving an average speedup of 3.23 compared to SparseIFDS while cutting the memory usage in half.

5.3.2 Results for RQ_2

To answer RQ_2, we measured the number of source facts that the analyses propagated for each procedure. Figure 6 shows a histogram representing this information per IFDS variant. The number of source facts is equal for bottom-up IFDS and MonoIFDS, so we examine them together here. Based on the results, we can see that the maximum number of source facts per procedure is 269 for MonoIFDS– way below our defined threshold of 384, qualifying our complexity assumption to hold on the complete benchmark. With MonoIFDS, in 92.6% of all analyzed procedures the number of source facts was even ≤ 57.

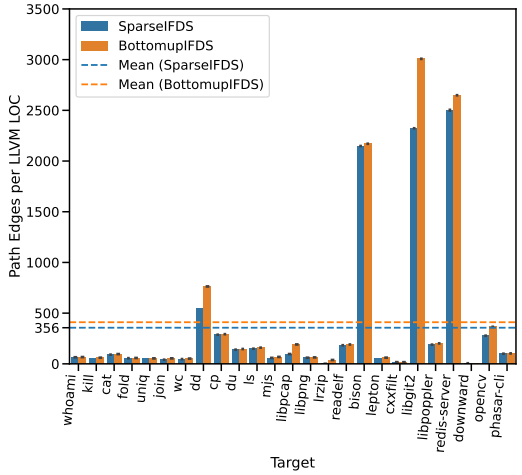


Figure 5 The number of path edges that SparseIFDS and the bottom-up IFDS variants need to propagate, normalized by the number of LLVM instructions of the respective targets.

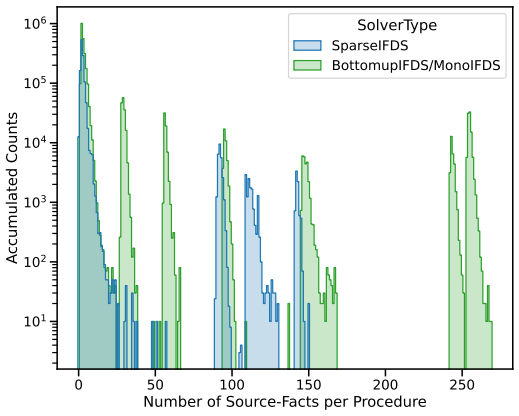


Figure 6 Histogram showing the number of source facts required per procedure for SparseIFDS and the bottom-up IFDS variants.

To answer RQ_2: In our entire benchmark the number of source facts per procedure is small enough to treat as constant, such that one can reasonably consider the effective runtime- and memory complexity of MonoIFDS to drop from the worst-case complexity to $\mathcal{O}(|N| \cdot |D|^2)$ and $\mathcal{O}(|N| \cdot |D|)$ respectively. In 92.6% of all measured procedures one could even reduce the threshold to less than a sixth without sacrifycing the conclusion.

5.3.3 Results for RQ_3

In the following we assess the impact of the discussed optimizations on the performance of MonoIFDS analyses, starting with the set-type for source facts. Figure 7 (left) shows the speedup of our default configuration for MonoIFDS using bitsets to store source facts, compared to a variant that uses hashsets instead. For all targets the runtime regressed with hashsets, with an average slowdown of 4.21 ± 1.90 and a maximum of 11.2 (for `redis-server`). Hence, taking the average speedup of 3.23 for MonoIFDS in comparison to SparseIFDS into account, one can attribute most of the observed speedup to using bitsets for representing the source-fact sets. For `Asterisk` and `Z3` the slowdown was large enough to exceed the time limit of 60 minutes, which the same analysis using bitsets did not. Also in terms of memory consumption, using hashsets is disadvantageous (see Figure 7 (right)), on average using 2.05 ± 1.45 times the memory required when using bitsets. For `OpenSSL` the MonoIFDS analysis even ran out-of-memory when using hashsets, which it did not when using bitsets. This shows that – as expected – using bitsets has a great impact on performance and should definitely be used in this case. However, one can also conclude that most of the speedup and memory savings of MonoIFDS seen in the evaluation of RQ_1 can be attributed to its use of bitsets instead of hashsets. Corollary, more design- and engineering effort is required when generalizing MonoIFDS to IDE problems, to avoid the expected slowdown of ∼4.21 and higher memory usage of $> 200\%$.

Considering the worklist iteration order, we compare RPO, a hybrid strategy, and our baseline using a simple deduplicated FIFO queue as worklist. Figure 8 shows that using RPO improves the runtime of MonoIFDS with an average speedup of 5.32 ± 6.92 and

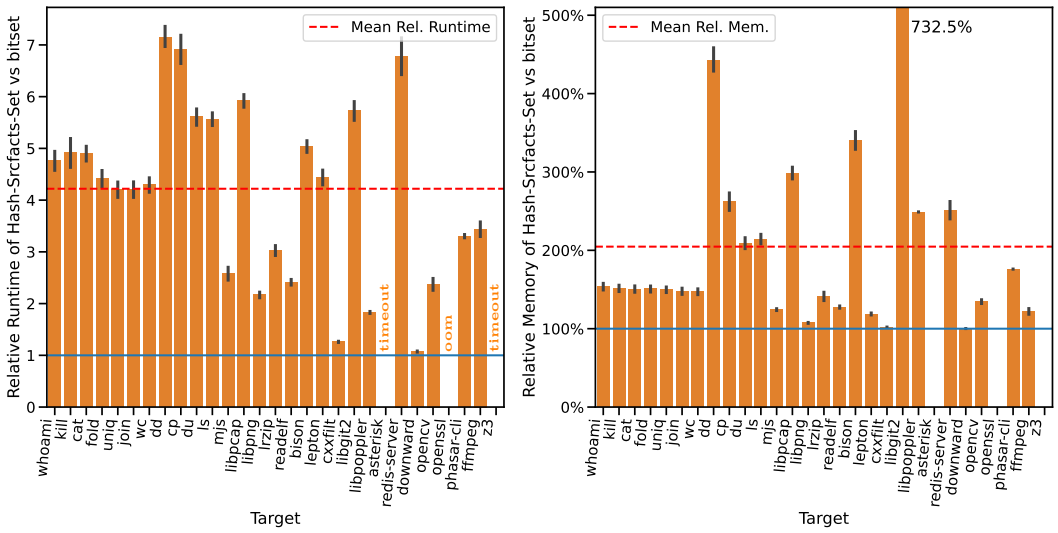


Figure 7 Performance regression of MonoIFDS when using hashsets instead of bitsets to represent the source fact sets. Left: Relative Runtime (1 means no slowdown, lower is better). Right: Relative memory consumption (lower is better). The labels *oom* and *timeout* denote that the respective MonoIFDS variant exceeded the given memory- or time limit.

maximum speedup of 33.0 for `cp`. However, for some programs, using RPO can actually make the analysis *slower*. Using the hybrid iteration strategy shows an average speedup of 4.96 ± 5.96 and a maximum speedup of 32.7 for `dd`; in no cases it performs worse than the RPO strategy, and only in a few cases slightly worse than the baseline. So, the hybrid strategy performs similar to RPO on average, while avoiding its pathological outliers. The performance differences corollate with the total number of worklist iterations needed to converge to a fixpoint, as shown by Figure 8 (right). While for many targets, MonoIFDS needed fewer iterations with RPO to finish its analysis, for some targets (especially the large ones) it requires *more* iterations, at worst 26× more for `redis-server`. Using the hybrid strategy, this problem can be mitigated, reaching the fixpoint with only $60.6\% \pm 26.4\%$ of the iterations when using a FIFO queue. However, the average number of iterations is with 177% for RPO higher than for the hybrid strategy (compared to the queue); so for *some* recursions the RPO strategy did still provide a benefit, although requiring more iterations. Comparing the observed number of worklist iterations with the measured running times, we see comparatively large variance for the running times that we do not see for the worklist iterations. So, we can attribute this variance to OS scheduling effects, and memory cache efficiency rather than to the amount of work to be done by the analysis algorithm.

Using the hybrid iteration strategy together with bitsets to represent the source facts, we measured an increased overall speedup of 6.17 on average compared to SparseIFDS.

MonoIFDS enables the successful integration of otherwise well-known optimizations into the IFDS algorithm. Using bitsets instead of hashsets to represent sets of source facts is a crucial optimization without drawbacks. For the worklist iteration strategy, in most cases the hybrid strategy provides most stable runtime performance with an average speedup of almost 5×, only in few cases outperformed by the FIFO queue.

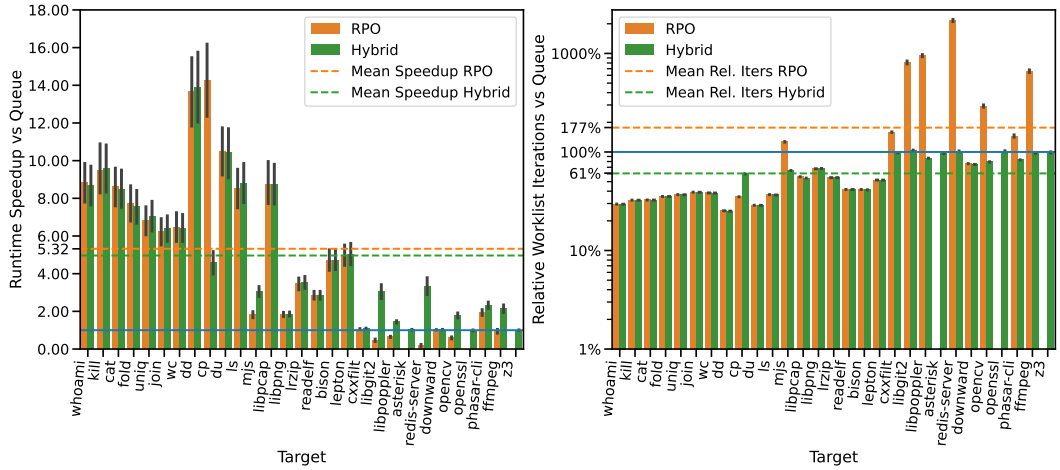


Figure 8 Performance improvements when optimizing the iteration strategy in MonoIFDS compared to using a FIFO queue. Left: Runtime speedup (higher is better). Right: Relative number of worklist iterations for reaching a fixpoint on all analyzed SCCs (lower is better, using a log-scale).

5.4 Discussion

In Figure 6 we have seen that the bottom-up variants of the IFDS algorithm need to generate more source facts at the beginning of the analyzed procedures. This can explain the slowdowns reported in Subsubsection 5.3.1 for bottom-up IFDS. When improving the heuristics from Subsection 3.1 to filter unnecessary source facts, this situation can probably be improved.

In Figure 8 we have also seen that the optimized worklist iteration strategy provides higher speedup for smaller programs than for larger programs. Manually investigating these larger programs reveals that for them the call graph contains a very large SCC (> 2000 procedures in some cases). Then the bottom-up analyses spend the majority of their time and memory analyzing this one large SCC, reducing the advantages of using a bottom-up analysis in the first place. One common source of spurious recursion that we found is due to the use of C++ virtual destructors. When PhASAR's call-graph analysis fails to resolve an indirect call via type information, it uses the LLVM function signature to match all address-taken functions against. Since all virtual destructors share the same LLVM function signature (partly due to opaque pointers[2]) `void(ptr %this)`, all of them need to be considered as potential callee targets. As destructors tend to call other destructors (to destroy owned sub-objects), this can easily lead to spurious recursion. We observe this behavior, e.g., for `OpenCV`. Using a more precise call-graph algorithm may improve the situation at the cost of potentially increased call-graph analysis time. Apart from performance implications, large CG-SCCs also impact the other benefits that we expected from a bottom-up analysis: When the analysis spends the majority of its time analyzing one big SCC, SCC-wise parallelization may be less effective; modular analysis that preserves analysis summaries per SCC may need to re-analyze the whole large SCC when only one procedure in it changes.

[2] Opaque pointers: `https://llvm.org/docs/OpaquePointers.html`

5.5 Threats to Validity

Internal Validity. The validity of our empirical evaluation may be threatened by various factors. The runtime measurements can be inaccurate due to CPU-boost or OS-scheduling. To mitigate this risk, we repeated the measurements 10 times and ensured that no unneeded process ran in parallel. Further, we instantiated the taint analysis only with a single configuration. The IFDS variants may perform differently for different taint configurations. To lower this risk, we carefully selected a real-world taint problem (CWE-415 double-free) that puts high load onto the solver, since *all* propagated data-flow facts are pointers, making this analysis heavily influenced by aliasing. This makes sure that the analysis is not "too easy", then falsely favoring a less scalable solution.

Our performance comparisons are only fair if all tested analysis variants compute the same results. To ensure this, we performed a not-measured analysis run and verified that all variants indeed compute the same taint leaks. However, for the four targets on which some of the analysis variants did not complete, either due to timeout or out-of-memory, a residual risk remains; given that the analysis variants computed the same results in all other cases, and that we shared parts of the implementation between the analysis solvers, we consider this risk acceptable.

We compared our novel MonoIFDS approach with a self-developed baseline; although we carefully optimized the baseline implementation, there remains a residual risk of biased results. To our knowledge, there is no other bottom-up IFDS analyzer for LLVM IR that we could reasonable compare against. In particular, we did not compare against Coyote [25], as it's main contribution lays in optimizing multi-core performance of bottom-up IFDS analyses, whereas MonoIFDS optimizes single-core performance.

External Validity. Our evaluation results may not generalize beyond our benchmarked set of target programs. To mitigate this risk, we reuse programs analyzed in other papers that deal with static analysis on LLVM IR. In addition, we make sure that we analyze a large and versatile set of target programs, selected to cover different sizes (∼5 000 to ∼5.2Mio LLVM LOC), languages (C and C++), and domains.

Furthermore, our evaluation has only been conducted on a single machine, which may threaten the generalization of our measurement results. To mitigate this risk, we refrained from enabling architecture-specific optimizations to our benchmark tool (via `-march=native`), instead we only relied on LLVM's generic `O2` pipeline.

6 Related Work

Bottom-up taint analysis has been described in the past already; in the following we list the most important related approaches.

Banerjee et al. [4] present CompTaint a compositional (i.e., bottom-up) taint analysis tool for Java and Python. Instead of relying on an analysis framework, such as IFDS, they integrate taint tracking into a bottom-up heap analysis. Our approach is completely decoupled from the concrete pointer/alias analysis and just expects an interface to query points-to or alias information from an external analysis that may or may not run upfront. Similar to our bottom-up analyses, they also embed sink-*effects* into their procedure summaries to defer reporting leaks at summary-application time; additionally, they also store source and sanitizer effects as sources and sanitizers may only trigger when, e.g., the receiver object has a certain type-state. Stievenart et al. [28] describe a bottom-up taint analysis for WebAssembly programs. In contrast to our work, they do not rely on IFDS but define the flow functions for

their analysis in an ad-hoc manner, similar to monotone frameworks. Instead of relying on a points-to or alias analysis to model the heap, they soundly but imprecisely represent heap memory by the special variable *mem*. Zhong et al. [35] present a bottom-up taint analysis for Java-based micro-services. They explicitly focus on supporting imprecise call graphs and therefore avoid computing call-graph SCCs (what most other bottom-up approaches including ours compute). Instead they compute procedure summaries in an arbitrary order, recursively visiting unanalyzed callee procedures on demand. They also do not rely on IFDS, explicitly mentioning its scalability problems, but build a context-sensitive and field-based *information propagation graph* per procedure to compute the procedure summaries. Our approach inherits context- and flow-sensitivity from IFDS and can be combined with any heap model, thus has the potential to deliver higher precision results, but – as shown in Subsection 5.4 – is sensitive to call-graph precision. It may be interesting to investigate how both approaches can be combined beneficially.

Shi et al. [25] present Coyote, a bottom-up IFDS taint analysis for LLVM IR that uses parallelization (through multi-threading) to tackle performance problems. Specifically, they partition procedures into smaller parts to increase the number of local analysis-tasks that can be executed in parallel, and to prevent that the analysis of a procedure must wait for the completion of all its callees' summaries. In contrast, our approach optimizes the single-threaded IFDS-based bottom-up analysis. The optimizations seem orthogonal and can likely be combined, especially one can integrate procedure partitioning into MonoIFDS. Finally, Choi et al. [7] describe STAR, an IFDS-based multi-source taint analysis tool for Android applications. Specifically, they introduce *symbolic summaries* to avoid redundantly propagating the same parameter fact for multiple taint-sources. Due to the symbolic facts, they can not always decide directly whether a sink may leak, as that may depend on the concrete source that has tainted the fact. For that they store so-called *effects* to defer reporting leaks at summary-application time. Our bottom-up approach also uses the sink-effects, but for a different purpose: to defer reporting leaks for facts that may never be generated. In contrast to our approach, STAR is not bottom-up, whereas our analyses do not propagate taint-sources. Hence, it may be interesting to investigate how to combine both approaches.

7 Limitations and Future Work

Despite significant performance improvements of MonoIFDS compared to state-of-the-art IFDS analysis approaches, there are some remaining limitations.

One issue lays in the data-flow-fact explosion that comes with field-sensitive analysis, e.g., using k-limited field-access paths. It is known that the number of data-flow facts scales exponentially with the allowed field-depth, which is a general problem and does not only apply to MonoIFDS. Although MonoIFDS handles large data-flow environments more efficiently than regular SparseIFDS, due to the exponential blowup it still cannot handle field-sensitive analyses with large k-limits efficiently. To address this, one could, e.g., integrate our optimizations from MonoIFDS into Synchronized Pushdown Systems [26] that promise to solve the fact-explosion issue. This, however, would constitute a major engineering effort.

The quality of strong updates (required to model sanitization in taint analysis) strongly depends on the used pointer analysis. Our evaluated implementation uses Steensgaard-style alias information, which are rather imprecise and, thus, severely limit the applicability of strong updates. To solve this, one could use a more precise pointer analysis, e.g., Boomerang [27], which to our knowledge has not been implemented for LLVM yet.

Finally, our presented definition of MonoIFDS does not support a precise handling of exceptions. To extend MonoIFDS to precisely support exceptions, one could split each procedure summary into two: one for normal returns, one for exceptional returns. The call-handling would need to be adapted (as it would for regular IFDS/IDE as well) to distinguish normal returns from unwind returns, and differentiate LLVM's call- and invoke-instructions (the latter is used to handle exceptions in LLVM IR). Since, to our knowledge, none of the popular frameworks PhASAR, Soot, SootUp, and Heros precisely support exceptions, and to limit the complexity of the MonoIFDS algorithm, we decided to not implement precise exception handling in this work.

8 Conclusion

In this paper we have presented an efficient approach to conduct a bottom-up analysis using a modified version of the IFDS algorithm to increase analysis performance and scalability. We have designed MonoIFDS, a bottom-up variant of the IFDS algorithm that overcomes known performance-drawbacks of a direct bottom-up formulation of SparseIFDS leveraging an optimized encoding of data-flows. We show that with MonoIFDS one can apply two well-known optimizations to the IFDS algorithm, bitset storage for sets and RPO worklist iteration order, which is not beneficial for SparseIFDS. We argue that with MonoIFDS the expected runtime complexity is far from the worst case and can reasonably be reduced by one factor of $|D|$. In our benchmark, we found the conditions for this complexity reduction fulfilled, on 92.6% of all analyzed procedures even with much stronger conditions.

Our evaluation shows that, although a basic bottom-up IFDS formulation frequently is *slower* than SparseIFDS, with MonoIFDS, one can achieve a speedup of 3.23 over SparseIFDS while only consuming 51% of its memory. We also found that a large amount of this speedup can be attributed to using bitsets instead of hashsets to represent source-fact sets. Since a generalization of MonoIFDS to Interprocedural Distributive Environment (IDE) problems would require the use of (hash-)maps, further research would need to show whether also a bottom-up variant of IDE could be implemented with similar efficiency nonetheless. Using the RPO worklist strategy has been found beneficial for analyzing programs with little (spurious) recursion in the call graph, which is why we developed a hybrid strategy using RPO only to analyze non-recursive procedures, with a FIFO-queue fallback for all other SCCs. We found this strategy to deliver the benefits of RPO when applicable, while avoiding large performance regression otherwise.

However, we also found that the benefits of MonoIFDS heavily depend on the quality of the used call graph: If the call graph contains too much spurious recursion, the expected benefits of a bottom-up analysis decline. Nonetheless, when combining hybrid RPO with MonoIFDS in our experiments we achieved an average overall speedup of 6.17 compared to SparseIFDS, highlighting the advantages of bottom-up analysis even with imprecise call-graph information, allowing one to target larger programs to analyze on consumer hardware than one could before.

References

1 Lars Ole Andersen. Program analysis and specialization for the c programming language. *PhD Thesis, University of Copenhagen*, 1994.

2 Steven Arzt. Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 1098–1110. IEEE, 2021. doi:10.1109/ICSE43902.2021.00102.

3 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 259–269. ACM, 2014. doi:10.1145/2594291.2594299.

4 Subarno Banerjee, Siwei Cui, Michael Emmi, Antonio Filieri, Liana Hadarean, Peixuan Li, Linghui Luo, Goran Piskachev, Nicolás Rosner, Aritra Sengupta, Omer Tripp, and Jingbo Wang. Compositional Taint Analysis for Enforcing Security Policies at Scale. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 1985–1996. ACM, 2023. doi:10.1145/3611643.3613889.

5 Eric Bodden. Inter-Procedural Data-Flow Analysis with IFDS/IDE and Soot. In *Proc. Int. Workshop on State Of the Art in Java Program Analysis (SOAP)*, pages 3–8. ACM, 2012. doi:10.1145/2259051.2259052.

6 François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications*, volume 735, pages 128–141. Springer-Verlag, 1993. doi:10.1007/BFB0039704.

7 Wontae Choi, Jayanthkumar Kannan, and Domagoj Babic. A scalable, flow-and-context-sensitive taint analysis of android applications. *J. Comput. Lang.*, 51:1–14, 2019. doi:10.1016/J.JVLC.2018.10.005.

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

9 Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Proc. Int. Conf. on Compiler Construction (CC)*, pages 159–179. Springer-Verlag, 2002. doi:10.1007/3-540-45937-5_13.

10 Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 567–577. ACM, 2011. doi:10.1145/1993498.1993565.

11 Dongjie He, Yujiang Gui, Yaoqing Gao, and Jingling Xue. Reducing the Memory Footprint of IFDS-Based Data-Flow Analyses using Fine-Grained Garbage Collection. In *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*, pages 101–113. ACM, 2023. doi:10.1145/3597926.3598041.

12 Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis. In *Proc. Int. Conf. Automated Software Engineering (ASE)*, pages 267–279. IEEE, 2020. doi:10.1109/ASE.2019.00034.

13 John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta informatica*, 7(3):305–317, 1977. doi:10.1007/BF00290339.

14 Kadiray Karakaya and Eric Bodden. Two sparsification strategies for accelerating demand-driven pointer analysis. In *Proc. Int. Conf. Software Testing, Verification and Validation (ICST)*, pages 305–316. IEEE, 2023. doi:10.1109/ICST57152.2023.00036.

15 Kadiray Karakaya, Palaniappan Muthuraman, and Eric Bodden. Pick your call graphs well: On scaling ifds-based data-flow analyses. In *Proc. Int. Workshop on State Of the Art in Program Analysis (SOAP)*, pages 43–50. ACM, 2025. doi:10.1145/3735544.3735587.

16 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. Int. Symp. Code Generation and Optimization (CGO)*, pages 75–88. IEEE, 2004. doi:10.1109/CGO.2004.1281665.

17 Haofeng Li, Haining Meng, Hengjie Zheng, Liqing Cao, Jie Lu, Lian Li, and Lin Gao. Scaling up the IFDS algorithm with efficient disk-assisted computing. In *Proc. Int. Symp. Code Generation and Optimization (CGO)*, pages 236–247. IEEE, 2021. doi:10.1109/CGO51591.2021.9370311.

18 Nomair A Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical Extensions to the IFDS Algorithm. In *Proc. Int. Conf. on Compiler Construction (CC)*, pages 124–144. Springer-Verlag, 2010. doi:10.1007/978-3-642-11970-5_8.

19 M Pnueli and Micha Sharir. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications*, pages 189–234, 1981.

20 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 49–61. ACM, 1995. doi:10.1145/199448.199462.

21 Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2):131–170, 1996. doi:10.1016/0304-3975(96)00072-2.

22 Fabian Schiebel, Florian Sattler, Philipp Dominik Schubert, Sven Apel, and Eric Bodden. Scaling Interprocedural Static Data-Flow Analysis to Large C/C++ Applications: An Experience Report. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 36:1–36:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ECOOP.2024.36.

23 Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 393–410. Springer-Verlag, 2019. doi:10.1007/978-3-030-17465-1_22.

24 Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Lossless, persisted summarization of static callgraph, points-to and data-flow analysis. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 2:1–2:31. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECOOP.2021.2.

25 Qingkai Shi and Charles Zhang. Pipelining bottom-up data flow analysis. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 835–847. ACM, 2020. doi:10.1145/3377811.3380425.

26 Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *POPL*, 3, 2019. doi:10.1145/3290361.

27 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 22:1–22:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.22.

28 Quentin Stiévenart and Coen De Roover. Compositional Information Flow Analysis for WebAssembly Programs. In *Proc. Int. Conf. Source Code Analysis and Manipulation (SCAM)*, pages 13–24. IEEE, 2020. doi:10.1109/SCAM51674.2020.00007.

29 Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proc. Int. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 264–280. ACM, 2000. doi:10.1145/353171.353189.

30 Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1:146–160, 1972. doi:10.1137/0201010.

31 TJ. Watson Libraries for Analysis. URL: http://wala.sourceforge.net/.

32 Xizao Wang, Zhiqiang Zuo, Lei Bu, and Jianhua Zhao. Dstream: A streaming-based highly parallel IFDS framework. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 2488–2500. IEEE, 2023. doi:10.1109/ICSE48619.2023.00208.

33 Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 435–446. ACM, 2013. doi:10.1145/2491956.2462159.

34 Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 197–208. ACM, 2008. doi:10.1145/1328438.1328464.

35 Zexin Zhong, Jiangchao Liu, Diyu Wu, Peng Di, Yulei Sui, Alex X Liu, and John CS Lui. Scalable compositional static taint analysis for sensitive data tracing on industrial micro-services. In *Proc. Int. Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 110–121. IEEE, 2023. doi:10.1109/ICSE-SEIP58684.2023.00015.