Two Sparsification Strategies for Accelerating Demand-Driven Pointer Analysis

Kadiray Karakaya
Heinz Nixdorf Institute
Paderborn University
Paderborn, Germany
kadiray.karakaya@upb.de

Eric Bodden
Heinz Nixdorf Institute
Paderborn University & Fraunhofer IEM
Paderborn, Germany
eric.bodden@upb.de

***Abstract*—To resolve aliasing, precise program analyses rely on pointer analyses. Demand-driven pointer analysis seeks to be efficient by computing information only for variables on which a demand is raised, through a points-to or alias query. Yet, research has shown that when applied to large-scale programs even demand-driven analyses can become expensive in terms of memory and runtime. This paper thus investigates to what extent demand-driven pointer analysis can be accelerated further if being executed over a sparse control-flow graph (CFG), specialized to those queries. We investigate two designs: First, type-aware sparsification, in which the resulting CFG only consists of statements containing variables that are type compatible with the query variable. Second, alias-aware sparsification, where the resulting CFG consists of the def-use chains of the query variable and all its intra-procedural aliases.**

We implement both designs in SPARSEBOOMERANG by extending BOOMERANG, a pointer analysis framework based on push-down systems. We evaluate SPARSEBOOMERANG by comparing it to BOOMERANG in terms of precision and performance. On the POINTERBENCH micro-benchmark suite for alias analysis, SPARSEBOOMERANG maintains the precision of BOOMERANG, in both designs. We evaluate the runtime and memory performance of SPARSEBOOMERANG by using FLOWDROID as a taint analysis client on real-world apps. Compared to the baseline BOOMERANG, on average SPARSEBOOMERANG solves alias queries 2.4x faster when using the type-aware sparsification strategy, and 2.8x faster when using the alias-aware variant with negligible memory overhead.

***Index Terms*—sparse pointer analysis, demand-driven analysis, data-flow analysis**

I. INTRODUCTION

Static program analysis clients are used in diverse application domains, including compiler optimization [1], bug [2] and vulnerability detection [3], and feature-based classification [4]. Client analyses are tailored depending on the requirements of the domain. Yet, independent of their domain, all such client analyses require pointer information [5]. For that purpose, they sometimes implement a pointer analysis as part of the client analysis [6], but more frequently use a pre-existing pointer analysis [7]. Fast and precise pointer analysis is still an open challenge for large-scale programs. To be precise, pointer analyses track calling contexts, fields, and statements, but that can hinder scalability. To be more scalable, many current pointer analyses are performed in a demand-driven manner [8], as opposed to conducting an exhaustive whole-program analysis [9]. They benefit from the fact that client analyses frequently require pointer information only for certain variables at certain program points. For instance, assume a direct assignment in a taint analysis, e.g., `x.f = t` where `t` is tainted. Here, aliases of `x` need to be known to the taint analysis so that this analysis can taint the `f` fields of `x`'s aliases, too. Demand-driven pointer analyses exploit just that: they compute alias information only for variables on which clients raise a demand through a query. Yet, previous work has shown that even demand-driven analyses can be expensive when run on large-scale programs [10].

BOOMERANG [10] is a state-of-the-art demand-driven pointer analysis framework that uses synchronized pushdown systems (SPDS) [11]. Pushdown systems (PDS) [12] are applicable to context-free language reachability problems, with which context-, and field-sensitivity can be modeled [11]. BOOMERANG synchronizes two PDS that model context- and field-sensitivity respectively. Both PDS depend on *rules* that correspond to data-flow functions. In this work, we exploit that many of these rules are *redundant* as they only affect data-flow facts that do not matter to the end result. Data-flow facts in pointer analysis correspond to the variables and their aliases. Redundant rules exist because control flow graphs (CFG) not only contain statements that affect the alias relationships, but also many other statements that do not. The beauty of demand-driven pointer analysis is that one knows the exact *query variable* (i.e., the variable, whose aliases are being queried), ahead of the analysis time. Therefore, when answering a raised demand one can sparsify the CFG by removing the statements that are irrelevant to the result *for the particular query variable*, and thus omit the redundant rules during the construction of the SPDS.

Previous work has successfully applied sparsification to improve the scalability of general data-flow analyses [13]–[15] and pointer analysis in particular [16]–[18]. All these approaches create sparse versions of the CFGs of a target program. These sparse versions are often called sparse value flow graphs (SVFGs) [14], or sparse control flow graphs (SCFGs) [19]. Most previous approaches create those SCFGs in a pre-analysis stage, for the whole program, and thus settle for the information available at that stage. Recent work by He et al. [19] showed that one can increase sparseness, i.e., omit from the CFG more irrelevant statements, by specializing

the SCFGs to the individual data-flow facts. Their work was applied to the IFDS [20] framework, which is applicable only to distributive analysis problems. Pointer analysis is known to be non-distributive [21]. In this work, we thus investigate to what extent one can make use of the idea of fact-specific sparseness nonetheless also in pointer analysis. In the proposed framework SPARSEBOOMERANG, the analysis creates a new SCFG specific to any queried value.

The goal of sparsification is to speed up the analysis run by restricting it to fewer program statements, while ideally generating results identical to those of an exhaustive analysis. Yet, the creation of the SCFGs itself incurs a cost both in terms of memory and runtime. Sparsification pays off when the savings during evaluating the sparse graph, in comparison to the original exhaustive graph, outweigh the construction time. To investigate this performance trade-off, in this work we present two sparsification strategies with different degrees of sparsification. Both strategies create on-demand SCFGs specific to each alias query. First, *type-aware sparsification* (TAS), where the resulting CFG only consists of the statements containing variables that are type compatible with the query variable. Second, *alias-aware sparsification* (AAS), where the resulting CFG consists of the def-use chains of the query variable and all its intra-procedural aliases. Those strategies mirror designs published earlier in the context of virtual call resolution [22], declared-type analysis (DTA) and variable-type analysis (VTA), respectively. DTA and VTA create assignment chains, where each node represents a variable either as its declared type (in DTA) or as itself (in VTA). Our strategies create def-use chains, where each node represents a statement either with the types (in TAS) or with the variables (in AAS) it contains.

We evaluate the applicability of the proposed two sparsification strategies within the SPDS framework. For that, we implement SPARSEBOOMERANG by extending the SPDS-based BOOMERANG. To validate whether the two sparsification strategies maintain the precision of the original exhaustive BOOMERANG, we run all approaches on the POINTERBENCH [23] benchmark suite for alias analysis. To evaluate the performance impact of the strategies, we run both BOOMERANG and SPARSEBOOMERANG on real-world Android applications. To this end, we extend FLOWDROID, a state-of-the-art taint analysis client for Android applications, so that it creates on-demand alias queries to BOOMERANG and SPARSEBOOMERANG. Evaluation results show that SPARSEBOOMERANG using either of the sparsification strategies solves the alias queries on average twice as faster as BOOMERANG, and while maintaining full precision. The performance gains achieved by the demand-driven pointer analysis are reflected in the taint analysis client, FLOWDROID.

To summarize, this paper presents these original contributions:

- Two sparsification strategies; type-aware sparsification and alias-aware sparsification for demand-driven pointer analysis,
- a sparse implementation of BOOMERANG, which we call SPARSEBOOMERANG, which maintains BOOMERANG's precision, and
- a modification of FLOWDROID that uses demand-driven pointer analyses BOOMERANG and SPARSE-BOOMERANG, and their performance evaluation on real-world android apps.

The remainder of the paper is organized as follows. In Section II, we present the background that our work is based on. In Section III, we introduce our on-demand sparsification strategies. In Section IV, we explain the implementation details of SPARSEBOOMERANG. Section V, presents the results obtained from our evaluations. In Section VI, we discuss the limitations of our approach and threats to its validity. In Section VII, we discuss the related work and we conclude with SectionVIII.

II. BACKGROUND

In this section, we introduce the key concepts that are required to understand the rest of the paper. First, we introduce sparse data-flow analysis. Then we explain how fact-specific sparsification works and why it is a good fit for demand-driven pointer analysis. We finally briefly explain BOOMERANG's approach to pointer analysis.

A. Sparse Data-flow Analysis

The precision of a data-flow analysis is determined by how accurately it reflects the effects of program statements on the data-flow facts. State-of-the-art precise data-flow analyses seek to be maximally context-, field-, and flow-sensitive. Context-sensitive analyses differentiate data-flow facts under different calling contexts. Field-sensitive analyses keep the different fields of a base object separate. Flow-sensitive analyses consider a program's control-flow ordering. Optimization approaches that aim to improve performance often compromise on these precision dimensions [24]–[27]. Sparsification, however, is a complementary optimization approach that aims to improve performance usually *without* sacrificing precision. Data-flow analyses can be instantiated to solve a wide range of analysis problems. Depending on the problem, flow functions, which model the effects of the program statements, differ. For instance, the effects of arithmetic operations are important for constant propagation analysis, but not for taint analysis. Sparsification approaches can speed up the analysis by instructing it to ignore statements that have no effect *in the context of that particular analysis problem*. CFG sparsification has been performed using def-use chains [18] and using SSA (static single-assignment) form [28], but often in a pre-analysis stage [17]. Such staged approaches are excessive for demand-driven analyses because they might sparsify program parts irrelevant to the raised demands.

B. Fact-specific Sparsification using Sparse IFDS

The IFDS (Interprocedural Finite Distributive Subset) framework by Reps et al. [20] is a widely adopted interprocedural data-flow analysis framework. While IFDS is not universally applicable: it requires the data-flow domain to be finite

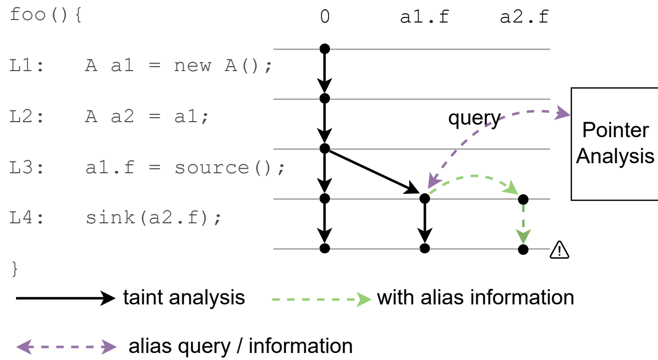


Fig. 1: Data-flow Graphs for Taint Analysis with Alias Information

and flow functions to be distributive over the meet operator, which is set-union. Nonetheless, IFDS has been instantiated for diverse analysis problems, including taint [6], [19], [29]–[31] and typestate analysis [32]–[34]. The IFDS algorithm represents data-flow problems as a graph-reachability problem, where each data-flow fact holds at a specific program point if its graph node is reachable at that point. Edges of the graph correspond to flow functions that reflect the effect of a statement on the data-flow facts.

Flow functions that do not affect *any* facts at a given statement are known as *id* functions. Recently, He et al. [19] observed that many non-*id* flow functions, in fact, behave as *fact-specific id* functions: while they may affect some data-flow facts, they are irrelevant to many others. Because IFDS, due to its distributivity, evaluates data-flow facts independent of each other, one can, during the evaluation of a data-flow fact d, safely disregard a flow function f if it is a d-specific identity function. Using this observation, He et al. introduced the Sparse IFDS algorithm, which includes fact-specific sparsification: it creates on-demand SCFGs *specific* to each data-flow fact that is being propagated. Facts are propagated to their next use point within their individual SCFGs. The original IFDS algorithm [20] instead propagates data-flow facts to *all* reachable program points. Fact-specific sparsification seems like an ideal fit for on-demand analyses because specific data-flow facts d only become known during analysis. He et al. exploited this during the demand-driven analysis of FLOWDROID, yet they did not assess sparsification in the broader context of pointer analysis, which is actually a non-distributive problem.

C. Demand-driven Pointer Analysis

Pointer analysis determines which program variables can point to which objects at runtime. It is required in real-world analysis problems where multiple program variables frequently point to the same object. Two variables that point to the same object are called aliases. Such alias information is crucial for a precise data-flow analysis for tracking the indirect data-flows through the aliases. Pointer analysis is usually not distributive at an assignment `x.f = t` one must assign aliases of `t` to the `f`-fields of *all* the aliases of `x`. For this reason, one cannot usually soundly handle all aliases independently, and the IFDS framework is not applicable, thus neither is Sparse IFDS.

Yet, as opposed to whole-program pointer analysis, demand-driven pointer analysis [8] is performed only for variables on which a demand, e.g., a pointer or alias query, is raised. It computes just enough information to satisfy the query. Interestingly, as Späth et al. showed [10], one can decompose a flow-sensitive pointer analysis such that when queries raise sub-queries at "points of indirection" (POI), e.g., at reads and writes to/from the heap, the evaluation of those sub-queries *does* become a distributive and thus distributively solvable analysis problem. Figure 1 shows the data-flow graphs that a taint analysis would produce with alias information. To know that the analysis must taint `a2.f` at `L3`, it must know that `a1` and `a2` alias at that point. In the case of a context- and flow-sensitive demand-driven pointer analysis, an alias query would look as follows:

$$Q(v, s, m)$$

v is the query variable for which alias information is required. s is the query statement and m its surrounding method. Thus, the *query* in Figure 1 would be instantiated as:

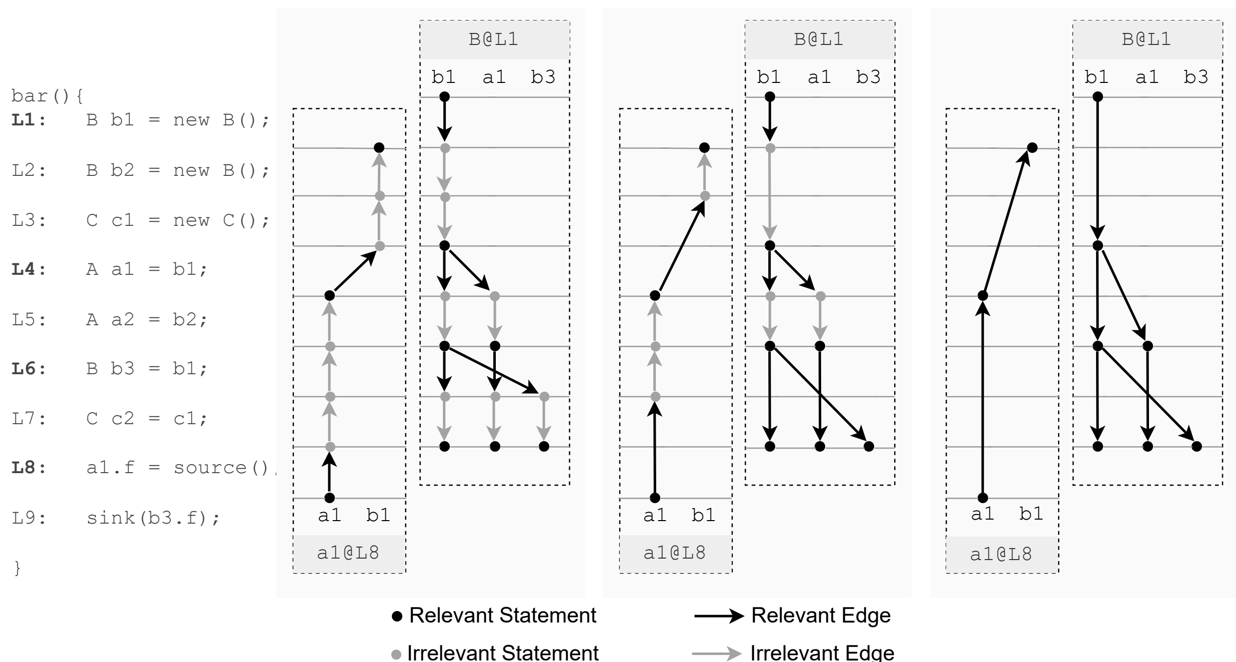
$$Q(a1, L3, foo())$$

In demand-driven pointer analysis, the query variable is used as the initial data-flow fact. It is provided explicitly, ahead of the analysis time. This allows one to perform on-demand fact-specific sparsification, building a SCFGs specific to each particular query.

D. BOOMERANG

Use of the IFDS framework requires data-flow functions to be distributive over the union meet operator [20]. However, precise pointer analysis is a non-distributive problem. With BOOMERANG, Späth et al. [10] showed that the pointer analysis problem can be modeled with distributive sub-queries that can still be solved with the IFDS algorithm. Such sub-queries are created at POIs. POIs cause the outer IFDS solver to instantiate sub-queries in the opposite direction, which are then again solved by inner IFDS solvers. BOOMERANG handles the following POIs:

- **Allocation Site:** Upon finding an allocation site (new object creation) during a backward analysis, a forward sub-query is created to find out which variables point to this object.
- **Field Write and Read:** Upon finding a field write statement during a forward analysis, a backward sub-query is created to find the aliases of the base variable of the field. Field read statements are handled in a backward analysis similar to the field write statements in the forward analysis.
- **Return and Call:** Return indirections are caused by the context change during the forward analysis and call indirections are caused during the backward analysis.



(a) Input for Pointer Analysis, where `B` is a subtype of `A` (b) Non-Sparse CFG (c) Type-Aware SCFG (d) Alias-Aware SCFG

Fig. 2: Data-flow Graphs of BOOMERANG's Analysis on Non-Sparse CFG and SPARSEBOOMERANG's Analyses on Type-Aware and Alias-Aware SCFGs

Recently, BOOMERANG has been reimplemented with SPDS [11] instead of the IFDS framework. IFDS and PDS are equally expressive and can be used to model the same inter-procedural data-flow problems [12]. SPDS uses two pushdown systems, Call-PDS and Field-PDS. The Call-PDS models flow- and context-sensitive data-flow analysis. Its push rules correspond to call-flow functions of the IFDS framework, whereas its pop rules correspond to return-flow functions. Normal rules of the Call-PDS are equivalent to the intra-procedural flow functions of IFDS, i.e., normal-flow functions and call-to-return-flow functions. The Field-PDS models flow- and field-sensitive data-flow analysis. Push and pop rules of the Field-PDS represent field store and field load statements respectively, where its normal rules correspond to assignments. SPDS improves over the IFDS via its compact encoding of field-sensitivity with the Field-PDS [11]. Call-PDS and Field-PDS both benefit from the proposed sparsification strategies because they both process the same CFG for the same query variable. From the sparsification point of view, the underlying solver (IFDS-, or SPDS-based) does not directly matter because, in the end, they both compute the same data flows over the same CFGs.

III. DEMAND-DRIVEN SPARSIFICATION STRATEGIES

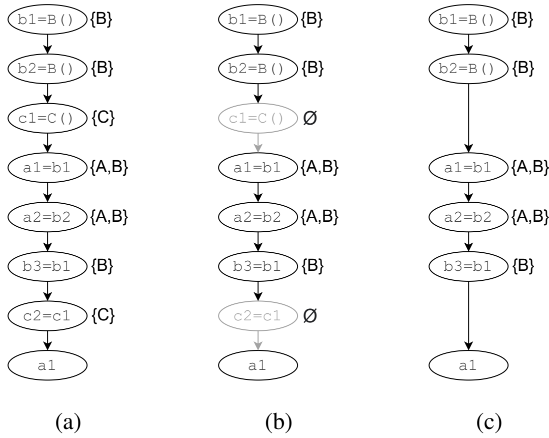
In Section II-C, we showed that an alias query consists of a query variable, a statement where its aliases are required, and a method that defines its context. Figure 2a shows an input program for pointer analysis. An example alias query looks as follows:

$$Q(a1, L8, bar())$$

We seek to find the aliases of `a1` at line `L8` in method `bar()`. Figure 2b shows how BOOMERANG performs this on a non-sparse CFG by default. It first initiates a backward pass (for `a1@L8`) to find the allocation site of the object, that `a1` points to. After finding the allocation site at `L1`, a POI, a forward pass for `B@L1` is initiated. The forward pass continues until it reaches the initial query location, yielding all variables that point to the same object (`B@L1`) as the query variable.

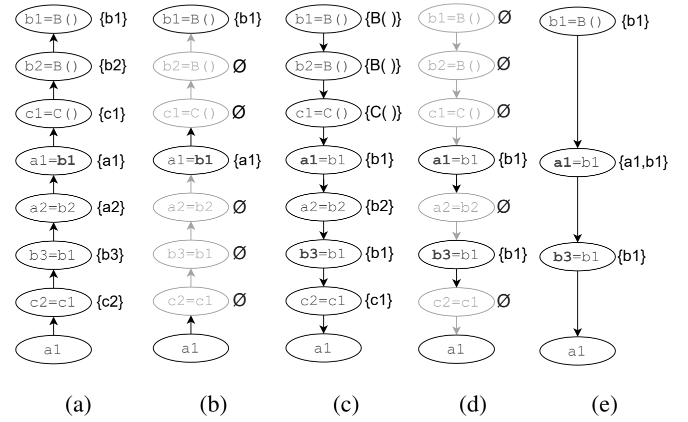
In fact, only the statements in lines `L1`, `L4`, `L6` can affect the aliasing relationships of `a1`, where `L8` is the query statement. Therefore, the edges that originate from the other statements are irrelevant. Irrelevant statements and edges that start from them are highlighted in Figure 2. Note that after sparsification, *relevant edges* connect to the *relevant statements* that are next in the respective SCFGs. By sparsifying CFGs, i.e., removing the redundant transition rules, a great deal of computation time can potentially be saved. However, sparsification consumes computation time as well, which depends on the degree of sparsification. Therefore, in the remainder of this paper we will seek to validate or refute the following assumption:

> **Assumption**: *A fine-grained SCFG is cheap to analyze, yet expensive to build, whereas a coarse SCFG is cheap to build, yet more expensive to analyze.*



(a) Each statement s_i is associated with its $var_types(s_i)$
(b) After applying $hierarchy_types(a1) \cap var_types(s_i)$
(c) The final $SCFG_{a1,L8,bar}$ after removing irrelevant statements

Fig. 3: Steps involved in Type-aware Sparsification



(a) Backward pass, each s_i is labeled with $uses(s_i)$, i.e., LHS
(b) After applying $intra_aliases(a1) \cap uses(s_i)$
(c) Forward pass, each s_i is represented by $uses(s_i)$, i.e., RHS
(d) After applying $intra_aliases(a1) \cap uses(s_i)$
(e) The final $SCFG_{a1,L8,bar}$, union of the relevant statements in all passes

Fig. 4: Steps involved in Alias-aware Sparsification

To investigate whether this assumption holds, we implemented two sparsification strategies with different degrees of sparsification. We will present these next.

Given a query, $Q(v, s, m)$, the sparse CFG specific to the query, $SCFG_{v,s,m}$, is obtained from the original CFG of the method m, CFG_m, by removing those statements that are irrelevant to the aliasing of v at the statement s. Below, we explain how the two sparsification strategies identify irrelevant statements.

A. Type-Aware Sparsification

Type-aware sparsification follows a heuristic that is inspired by Declared-Type Analysis [22], which is based on the following idea. Given a program written in a strongly typed language, a variable can only point to an object compatible with its declared type. In object-oriented languages such as Java, this definition includes any types that are subtypes or supertypes of the declared type of the variable. Supertypes need to be included to incorporate the possibility of explicit casts. Accordingly, an assignment, leading to aliasing, can only happen between two variables whose types are in a subtype-supertype relationship. Therefore, given a query variable, one can obtain an SCFG by keeping only such relevant statements. Type-aware sparsification retains the relevant statements as follows. Given a query variable v, $hierarchy_types(v)$ is the set of types in the type hierarchy of v's declared type, i.e., its sub- and supertypes. $var_types(s)$ is the set of types that the statement s references, e.g., the type of the left-hand side and right-hand side for an assignment, or the argument types and base type for a method call. Then, s is a *relevant statement with respect to* v if and only if:

$$hierarchy_types(v) \cap var_types(s) \neq \emptyset$$

Figure 3 shows how type-aware sparsification works for the query $Q(a1, L8, bar())$ in the example program in Figure 2a. The type `B` is a subtype of `A`, therefore $hierarchy_types(a1) = \{A, B\}$. Figure 2c shows how SPARSEBOOMERANG solves an alias query over the resulting $SCFG_{a1,L8,bar}$. Note that it still contains irrelevant edges, due to the coarse-grained type-aware approach.

B. Alias-Aware Sparsification

Alias-aware sparsification likewise follows the approach introduced with the VTA algorithm [22], it represents variables by themselves, here denoted by variable names. VTA uses def-use chains. Definitions and uses of a variable cause aliasing. Intuitively, one can obtain an SCFG that only consists of the statements that belong to the def-use chain of the query variable. However, it is also necessary to be aware of the def-use chains of all the aliases created in the initial def-use chain, until a fixed point is reached where there are no new aliases to be discovered. To ensure this, alias-aware sparsification works in two passes, similarly to BOOMERANG [10] but intra-procedurally. First, a backward pass is performed until an allocation site is found, then a forward pass follows until the query statement is reached. There may be multiple such passes, whose details are explained in Section IV-B. Alias-aware sparsification retains the relevant statements as follows. Given a query variable v, $intra_aliases(v)$ is the set of intra-procedural aliases of v. $uses(s)$ is the set of variables used in the statement s, then s is a *relevant statement with respect to* v if and only if:

$$intra_aliases(v) \cap uses(s) \neq \emptyset$$

We maintain $intra_aliases(v)$, which initially only contains the *v* itself. The meaning of *use* depends on the direction of the pass. For instance, in the backward pass, the left-hand side (LHS) of an assignment is in the $uses(s_i)$ and in the

TABLE I: Statements Handled by Type-aware Sparsification

Statement	IR	var_types	Effect on typeWorklist
assign	$x \leftarrow y$	$\{t(x), t(y)\}$	—
cast	$x \leftarrow (T)y$	$\{t(x), t(y)\}$	—
load	$x \leftarrow y.f$	$\{t(x), t(f)\}$	$add(t(y))$
store	$x.f \leftarrow y$	$\{t(f), t(y)\}$	$add(t(x))$
invoke	$r \leftarrow b.m(a_i.f)$	$\{t(r), t(f)\}$	$add(t(b), t(a_i))$

forward pass the right-hand side (RHS). Accordingly, in the backward pass, the RHS of an assignment is added to the set $intra_aliases(v)$ and in the forward pass the LHS.

Figure 4 shows how alias-aware sparsification works for the query $Q(a1, L8, bar())$ in the example program in Figure 2a. In the backward pass, initially $intra_aliases(a1)$ is $\{a1\}$, after `a1=b1` it becomes $\{a1, b1\}$. In the forward pass, initially $intra_aliases(a1)$ is $\{b1\}$, after `a1=b1` it becomes $\{b1, a1\}$ and after `b3=b1` it becomes $\{b1, a1, b3\}$. The resulting $SCFG_{a1,L8,bar}$ contains the union of the relevant statements from both passes. Figure 2d shows how SPARSEBOOMERANG solves an alias query over $SCFG_{a1,L8,bar}$ that is obtained via the alias-aware sparsification.

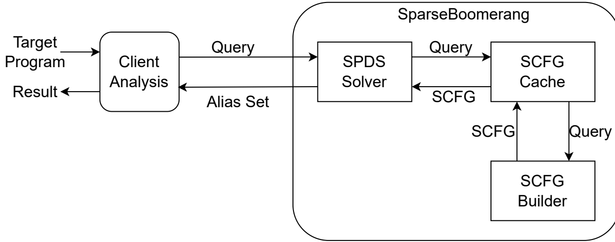


Fig. 5: System Overview of SPARSEBOOMERANG

IV. SPARSEBOOMERANG

In this section, we explain the implementation details of our approach. Figure 5 shows the system overview of SPARSE-BOOMERANG. It applies a caching mechanism similar to that of Sparse IFDS [19], with a nuance that SCFGs are cached per query instead of per data-flow fact. Queries can be both originating from the client or internal queries that the SPDS solver issues, e.g., on switching contexts. Depending on the configured sparsification strategy (type-aware or alias-aware), the corresponding SCFG builder and the cache are instantiated.

In Section III, we explained how the proposed sparsification strategies find *relevant statements*. We next explain how the statements are handled at the intermediate representation (IR) level, and introduce the algorithms for each strategy.

A. Implementation of Type-Aware Sparsification

Table I shows the statements handled by type-aware sparsification with their IR. *assign* and *cast* statements are handled the same, but we make the distinction to point out that assignments from both supertypes and subtypes exist. *load* and *store* statements concern reading from, and writing to the heap using field references. This makes it necessary to track the aliases of their base variables. To do so, we maintain a worklist of types, *typeWorklist*, where we store the declared types of the base variables of field references and process them in the subsequent iterations. *var_types* correspond to the declared types of the variables involved in a statement. Note that *invoke* statements may have multiple arguments (e.g., $b.m(a_1.f, a_2.f, ...)$), so each of them must be included.

Algorithm 1 shows how type-aware sparsification works. It takes as input the variables passed as part of the alias query, $Q(v, s, m)$. *relevantStmts* is the set of statements that are relevant to the alias query. *typeWorklist* is initiated with the type of the query variable, *type(v)*. The algorithm works until the *typeWorklist* is empty, e.g., there are no further relevant types to process.

```
1  Function TypeAwareSparsification(v, s, m):
2      relevantStmts ← {}
3      typeWorklist ← {type(v)}
4      while typeWorklist ≠ {} do
5          Get t from typeWorklist
6          FindRelevantStmts(t, s, m)
7      end
8      Sparsify(m, relevantStmts)
9  Function FindRelevantStmts(t, s, m):
10     foreach s_i ≪ s in m do
11         if var_types(s_i) ∩
               hierarchy_types(t) ≠ ∅ then
12             Add s_i to relevantStmts
13             HandleBase(s_i)
14         end
15     end
16 Function HandleBase(s_i):
17     if s_i contains field then
18         Add type(base(field)) to typeWorklist
19     end
```

Algorithm 1: The Algorithm of Type-Aware Sparsification

FindRelevantStmts iterates over the CFG of the query method *m* until it reaches the query statement *s*. The method *HandleBase* identifies the statements that contain a field reference and populates the *typeWorklist* with the type of their base variables, i.e., *type(base(field))*. To *sparsify* the CFG, it is traversed at the end to retain the *relevantStmts*.

B. Implementation of Alias-aware Sparsification

Table II shows the statements handled by alias-aware sparsification. In this case, we additionally handle *allocation* and *identity* statements, which were treated as simple assignments by the type-aware variant. *allocation* indicates that a variable is instantiated within the current CFG. *identity* denotes a mapping from a method argument to a local variable. *identity* and *load* indicate that a variable is instantiated elsewhere. *store* signals that the base variable of the field reference must be handled. *invoke* is a special case. In a backward pass, it must be handled similarly to an allocation, in a forward pass it must be handled similarly to a field store. As explained in

Section III-B, the meaning of *uses* depends on the direction of the pass, therefore we denote the uses for the backward pass as *bw_uses* and for the forward *fw_uses*. To discover the aliasing relationships caused by points of indirections (POI) [10], in this strategy, we maintain two worklists. A backward worklist is used to create the def-use chains in the backward pass, and a forward worklist is used to create them in the forward pass. POIs can be discovered during each pass. POIs discovered in a backward pass cause *bw_uses* to be added to the forward worklist. POIs discovered in a forward pass cause base variables or invocation receivers to be added to the backward worklist.

TABLE II: Statements Handled by Alias-aware Sparsification

Statement	IR	bw_uses	fw_uses	POI for
assign	$x \leftarrow y$	$\{x\}$	$\{y\}$	—
cast	$x \leftarrow (T)y$	$\{x\}$	$\{y\}$	—
allocation	$x \leftarrow T()$	$\{x\}$	—	Backward
identity	$x \leftarrow arg$	$\{x\}$	—	Backward
load	$x \leftarrow y.f$	$\{x\}$	$\{f\}$	Backward
store	$x.f \leftarrow y$	$\{f\}$	$\{y\}$	Forward
invoke	$r \leftarrow b.m(a_1.f)$	$\{r\}$	$\{b, f\}$	Both

Algorithm 2, shows how alias-aware sparsification works. It may perform multiple backward and forward passes depending on the number of POIs. To be brief, we assume that the method *uses()* acts as *fw_uses* or *bw_uses*, depending on the direction of the pass. Similarly, we assume *intra_aliases(v)* is maintained implicitly. Both algorithms soundly preserve branching statements and stop processing after reaching the query statement.

V. Evaluation

Sparsification aims to improve the performance of the analyses, while still preserving their precision. Therefore, we evaluate the impact of the proposed sparsification strategies considering two dimensions: precision, and performance impact. To do so, we have formulated the following research questions:

- RQ1: Do the sparsification strategies cause precision loss?
- RQ2: How do the sparsification strategies impact the performance of the demand-driven pointer analysis and its client?
- RQ3: How does the degree of sparsification impact the SCFG construction time and its evaluation time?

A. Experimental Setup

SparseBoomerang, available at https://github.com/secure-software-engineering/SparseBoomerang, extends the latest version of Boomerang at the time of writing (`1179227`) [35]. We use FlowDroid as a taint analysis client with its default source and sink definitions. We also extended the latest version of FlowDroid (`d97f9d9`) [36] so that it creates on-demand alias queries for Boomerang and SparseBoomerang instead of using its own integrated alias analysis. Both tools are based on Soot static analysis

```
1  Function AliasAwareSparsification(v, s, m):
2      relevantStmts ← {}
3      bwWorklist ← {v}
4      fwWorklist ← {}
5      while bwWorklist ≠ {} do
6          Get b from bwWorklist
7          FindRelevantStmts(b, s, m)
8          while fwWorklist ≠ {} do
9              Get f from fwWorklist
10             FindRelevantStmts(f, s, m)
11         end
12     end
13     Sparsify(m, relevantStmts)
14 Function FindRelevantStmts(v, s, m):
15     foreach s_i ≪ s in m do
16         if uses(s_i) ∩ intra_aliases(v) ≠ ∅
            then
17             Add s_i to relevantStmts
18             HandlePOI(s_i)
19         end
20     end
21 Function HandlePOI(s_i):
22     if s_i is POI for backward then
23         Add uses(s_i) to bwWorklist
24     end
25     if s_i is POI for forward then
26         Add base(field) to fwWorklist
27     end
```

Algorithm 2: The Algorithm of Alias-Aware Sparsification

framework [37]. We use the following benchmarks in our experiments:

- **PointerBench**: PointerBench [23] is a micro-benchmark suite for alias analysis. We use this suite to evaluate the correctness of the sparsification approaches. We check whether we can obtain the same aliases by issuing alias queries to Boomerang without sparsification and to SparseBoomerang with type-aware and alias-aware sparsification strategies.
- **Real-world Apps**: We include real-world Android apps to investigate the performance of our approach under the workload of large-scale and complex programs. For that, we selected the 20 most downloaded Android apps from the Google Play store listed in androidrank.org [38], then we downloaded their most recent version from Androzoo [39].
- **Replication Package**: We provide a replication package that contains the complete toolchain to reproduce the findings, along with their source codes. The replication package is available at https://drive.google.com/drive/folders/1UTckUbqKs54Org3BBkba_SJssih6Ugds

All the experiments were performed on a Macbook Pro with a Quad-Core Intel i7 processor at 2,3 GHz and 32 GB memory.

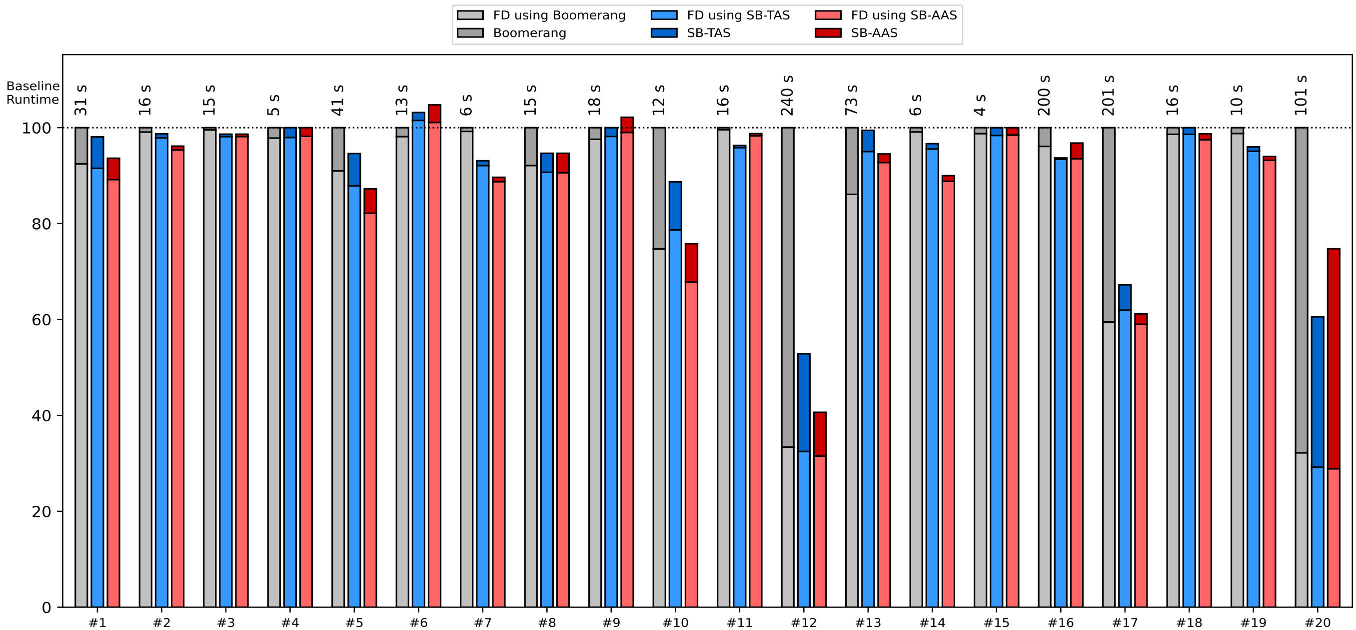


Fig. 6: Relative time spent on taint analysis and solving alias queries by FLOWDROID (FD) using baseline BOOMERANG, SPARSEBOOMERANG (SB) with TAS and with AAS, in %

The JVM was configured with a maximum heap size of 25GB, and a maximum stack size of 1GB. All performance data was generated as the average of five runs of each input app with each alias analysis.

B. RQ1: Do the sparsification strategies cause precision loss?

It is crucial for the sparsification approaches to maintain the precision of their non-sparse counterparts. Sparsification aims to reduce the number of statements that are irrelevant to the particular analysis, but this requires a careful study of the program statements to find out how to handle each one of these. We, therefore, test whether both approaches maintain the level of precision that is obtained by non-sparse BOOMERANG, on POINTERBENCH. The micro-benchmark suite contains 35 target programs. Its basic tests include branching, loops, recursion, and inter-procedural aliasing. It also includes corner cases where field-, flow-, and context-sensitivities are tested. The results show that both sparsification approaches report results identical to the non-sparse analysis. Precision, in particular, is therefore maintained.

C. RQ2: How do the sparsification strategies impact the performance of the demand-driven pointer analysis and its client?

As discussed in Section IV, both sparsification strategies come at a cost. They need to build, on-demand, sparse versions of the original CFGs of the input programs. To measure the impact of the sparsification strategies on solving alias queries, and on the overall runtime of the taint analysis client, we evaluate the performance of SPARSEBOOMERANG by comparing it to BOOMERANG.

Figure 6 shows the relative time spent by FLOWDROID on taint analysis and by BOOMERANG and SPARSEBOOMERANG on solving alias queries on each input app. FLOWDROID using BOOMERANG is used as the baseline. Each run by SPARSEBOOMERANG with TAS, and with AAS is normalized against this baseline. It can be observed that, despite their cost in SCFG construction, both sparsification strategies frequently reduce the time spent by demand-driven pointer analysis on solving alias queries. More specifically, SPARSEBOOMERANG, compared to BOOMERANG, solves the alias queries on average 2.4x faster with type-aware sparsification, and 2.8x faster with the alias-aware variant. The maximum speedups achieved by each strategy are 14.6x and 18.7x respectively. The speedups gained during the pointer analysis are also reflected in the client analysis. FLOWDROID using SPARSEBOOMERANG performs the taint analysis on average 1.13x faster with type-aware sparsification and 1.17x faster with alias-aware sparsification. The maximum speedups by each strategy are 1.9x and 2.5x respectively. The full set of absolute numbers is contained in Table III.

To investigate the significance of the results, we have also performed Wilcoxon signed-rank test [40] at 0.05 significance level. Both TAS (p=0.0027) and AAS (p=0.0094) improve the performance of the pointer analysis significantly. Similarly, the client's performance also increases significantly when using TAS (p=0.0012) and AAS (p=0.0011).

Figure 7 shows the maximum memory consumption of FLOWDROID using SPARSEBOOMERANG with TAS and AAS compared to the baseline memory consumption of FLOWDROID using BOOMERANG. On average, the maximum memory consumption increases. We have measured an average of 3% increase in memory consumption when using SPARSE-

TABLE III: Performance of FLOWDROID using the baseline BOOMERANG (B) and SPARSEBOOMERANG with TAS and AAS

#	APK	Runtime (s)					Memory (GB)			Total Query Time (ms)					Query Solv. (ms)			SCFG Const. (ms)		DoS	
		B	TAS	B/TAS	AAS	B/AAS	B	TAS	AAS	B	TAS	B/TAS	AAS	B/AAS	B	TAS	AAS	TAS	AAS	TAS	AAS
1	candycrush	31	30	1.02	29	1.07	2.33	2.06	1.82	2368	2062	1.15	1396	1.70	2368	2029	1357	32	39	0.30	0.33
2	chrome	15	15	1.01	15	1.04	0.69	0.69	0.71	141	130	1.08	126	1.12	141	113	111	17	15	0.40	0.51
3	excel	14	14	1.01	14	1.01	0.65	0.68	0.72	67	74	0.91	73	0.92	67	64	64	9	9	0.75	0.77
4	fblite	5	5	1.00	5	1.00	0.18	0.18	0.18	108	102	1.06	90	1.20	108	81	68	21	22	0.47	0.49
5	garena	40	38	1.06	35	1.15	1.48	1.12	0.99	3674	2740	1.34	2081	1.77	3674	2624	1885	116	196	0.37	0.42
6	gclock	12	13	0.97	13	0.95	0.72	0.70	0.77	233	209	1.12	463	0.50	233	172	439	37	23	0.43	0.53
7	gfiles	5	5	1.07	5	1.12	0.19	0.19	0.19	45	57	0.78	52	0.86	45	40	39	17	12	0.33	0.55
8	gkeyboard	15	14	1.06	14	1.06	0.96	0.74	0.84	1183	595	1.99	609	1.94	1183	414	446	181	162	0.33	0.52
9	gsearchlite	18	18	1.00	18	0.98	0.97	0.90	0.97	444	335	1.32	584	0.76	444	287	550	48	33	0.44	0.51
10	mivideo	12	11	1.13	9	1.32	0.32	0.80	0.78	3134	1240	2.53	992	3.16	3134	1085	884	155	108	0.52	0.53
11	msword	16	15	1.04	16	1.01	0.72	0.72	0.72	66	72	0.91	75	0.88	66	62	65	10	9	0.75	0.77
12	mxplayer	239	126	1.89	97	2.46	5.59	6.04	2.35	159583	48761	3.27	21874	7.30	159583	45194	15304	3567	6569	0.40	0.47
13	netflix	73	72	1.01	69	1.06	2.14	1.25	0.90	10150	3228	3.14	1306	7.77	10150	3125	1216	102	90	0.45	0.73
14	shareit	6	5	1.03	5	1.11	0.32	0.30	0.36	53	66	0.80	70	0.76	53	58	61	8	9	0.32	0.54
15	shareme	4	4	1.00	4	1.00	0.23	0.23	0.23	49	64	0.76	61	0.80	49	52	49	12	11	0.46	0.56
16	tiktok	199	187	1.07	193	1.03	3.56	2.36	5.56	7801	534	14.59	6501	1.20	7801	499	4760	35	1740	0.37	0.31
17	ucmobile	201	135	1.49	123	1.63	4.64	3.63	3.00	81620	10632	7.68	4374	18.66	81620	10137	3498	495	875	0.47	0.49
18	viber	15	15	1.00	15	1.01	0.74	0.74	0.70	215	215	1.00	190	1.13	215	199	175	16	15	0.39	0.44
19	webview	10	9	1.04	9	1.06	0.48	0.48	0.48	121	91	1.33	80	1.50	121	79	68	12	12	0.44	0.55
20	whatsapp	101	61	1.65	75	1.34	0.77	1.28	2.78	68743	31784	2.16	46522	1.48	68743	31658	46030	126	491	0.42	0.36

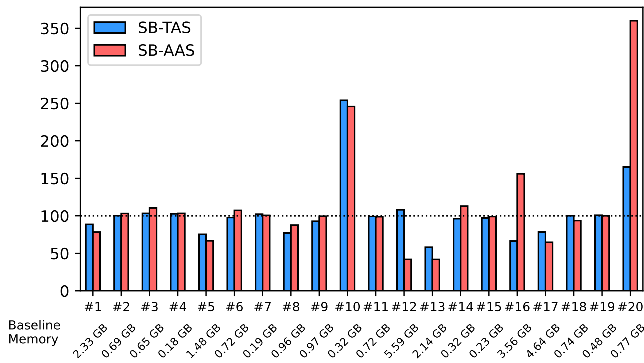


Fig. 7: Relative Memory Consumption of FLOWDROID using SPARSEBOOMERANG with TAS and with AAS compared to the baseline BOOMERANG, in %

BOOMERANG with TAS, and 13% when it's using AAS. However, according to the Wilcoxon signed-rank test, memory increases with TAS (p=0.24) and AAS (p=0.70) are insignificant. The impact on memory consumption is largest for apps #10, and #20. When observing the same apps in Figure 6, it can be observed that these apps also benefited from a large speedup in the analysis runtime.

An increase in memory consumption was expected. This is because, after all, both sparsification strategies make use of caching to reduce the amount of time spent on sparsification in case the same queries are issued. However, surprisingly, we see that for some subject apps, e.g., #5, #13, and #17, sparsification substantially *decreases* memory consumption. We attribute this to savings in the client analysis which, given the sparsification, needs to associate data-flow facts with fewer CFG nodes.

D. RQ3: How does the degree of sparsification impact the SCFG construction time and its evaluation time?

We have already informally used the term *degree of sparsification (DoS)*. We define it formally as follows. Given an input program p, M is the set of all the methods of p in which an alias query is issued. Let m a method in M, where CFG_m is its original non-sparse CFG, and $SCFG_m$ is its sparse SCFG. $|CFG_m|$ is the number of statements in CFG_m and $|SCFG_m|$ is the number of statements in $SCFG_m$. DoS_p is then calculated as:

$$DoS_p = \frac{\sum_{m \in M} |CFG_m| - |SCFG_m|}{\sum_{m \in M} |CFG_m|}$$

In Section III, we made the assumption that a higher DoS would lead to a larger decrease in runtime when solving alias queries. To investigate this, in Figure 8, we show, for each run, the correlation between DoS and the average time taken to solve alias queries in these runs. The trend shows the assumed inverse relation on a small scale. When the DoS increases, i.e., when more irrelevant statements are removed, it takes less time to solve the alias queries. Accordingly, when the DoS decreases, i.e., when it is necessary to retain a large fraction of relevant statements, on average it takes more time to solve the alias queries.

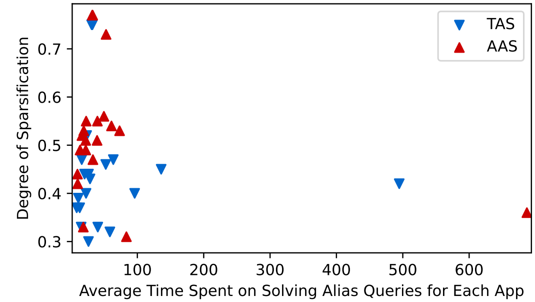


Fig. 8: Degree of sparsification (DoS) and average time spent on solving alias queries

To further highlight the impact of the DoS, in Figure 9 we show the relative time spent by each sparsification strategy on constructing the SCFGs, and then solving the alias queries over them. We observe that the assumption generally holds: in most cases, a higher degree of sparsification shortens the alias-query evaluation time. However, the results on apps #6, #7,

#8, #9 contradict the rule: counter-intuitively for those apps, we see that the construction of the type-aware SCFGs actually takes longer than the construction of the alias-aware SCFGs, although the latter actually operates on a more detailed data structure, the def-use chains.

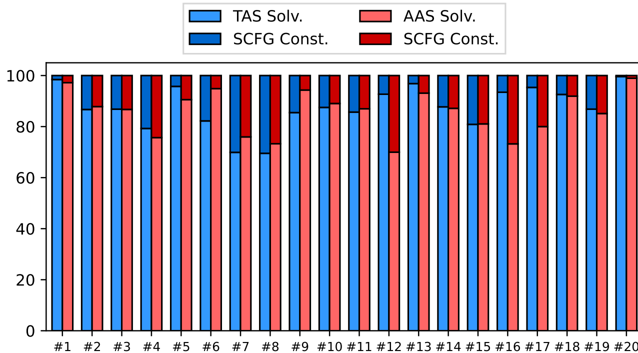


Fig. 9: Relative Time Spent on Solving the Alias Queries and Constructing the SCFGs by each Strategy, in %

We have preferred calculating DoS based on the number of statements, to have a common metric that can be used by both the pointer analysis and the sparsification approaches. While the complexity of pointer analysis depends on the number of edges in the CFG, the complexities of the sparsification strategies depend on the diversity of the base types (for TAS) and the number of POIs (for AAS).

VI. Threats to Validity

We have used the most installed apps on androidrank.org [38]. Among them, we have discarded the ones that did not contain any sources or sinks that FlowDroid could detect with its default configuration. Further, we have ignored the apps that caused an error for the underlying static analysis framework, Soot. This might introduce some selection bias, however, appears hard to avoid.

To even out noise in runtime and memory measurements, we measured five runs and here report the average over these five runs.

Because both the pointer and the client analysis are running in the same process, it is very hard to attribute increases or decreases in memory consumption to either of them, let alone individual data structures and algorithms. We thus focus on reporting the overall consumption.

VII. Related Work

Sparsification has been applied in diverse static-analysis settings. Pinpoint [13] is a staged sparsification approach that uses intraprocedural data dependence to selectively solve only the necessary interprocedural data dependence queries. SVF [14] uses as input points-to information that is generated by a cheap imprecise analysis, constructs value-flows which are then used for a precise sparse analysis. Sparsification approaches are usually specific to particular analyses, yet Oh et al. [15] introduced a general sparsification framework that is theoretically applicable to any analysis. Our work can be seen as an instantiation of their framework.

Applications of sparsification have also been performed for pointer analysis in particular. SFS [17] is a flow-sensitive pointer-analysis approach that uses sparse def-use chains created by a flow-insensitive analysis stage. With alias-aware sparsification, we create def-use chains too, except in a demand-driven manner utilizing the query information available at the analysis time. Hardekopf and Lin [18] introduced a semi-sparse approach, where sparsification is only applied to top-level variables. This approach could be employed as a further, more coarse-grained sparsification strategy. SPAS [16] is a path-sensitive sparsification approach that is applied in stages to pointer analysis. Handling path-sensitivity is beyond the scope of our study.

Many of the existing approaches sparsify program parts in a pre-analysis stage, where only limited information about the target program is available. The Sparse IFDS algorithm by He et al. showed that further sparsification is possible when applying sparsification on-demand and using the information available at the runtime of the analysis. Their approach is also demonstrated by extending FlowDroid. Yet, a direct comparison with their approach was not possible because they sparsify the FlowDroid itself whereas we sparsify Boomerang. So the impact of our approach is only indirectly reflected in FlowDroid. FlowDroid is multithreaded, but Boomerang currently does not support multithreading, so neither does SparseBoomerang.

VIII. Conclusion and Future Work

In this work, we proposed two sparsification strategies to accelerate demand-driven pointer analysis. Both strategies create query-specific sparse CFGs by utilizing the information available at the analysis runtime. Although sparse CFG construction takes up time, we have shown that it is negligible given the achieved speedups. The proposed strategies soundly preserve branching statements, further sparsification can be achieved by identifying where branching statements themselves become irrelevant due to the removal of other statements. The proposed strategies have been applied to pointer analysis, but in the future, we plan to investigate their applicability also to general demand-driven context- and flow-sensitive data-flow analysis problems. In this study, we presented a comparison of the two strategies. However, in the future, we also seek to apply a combination of both strategies.

References

[1] G. A. Kildall, “A unified approach to global program optimization,” ser. POPL ’73. New York, NY, USA: Association for Computing Machinery, 1973, p. 194–206. [Online]. Available: https://doi.org/10.1145/512927.512945

[2] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, “Using static analysis to find bugs,” *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[3] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: a static analysis tool for detecting web application vulnerabilities,” in *2006 IEEE Symposium on Security and Privacy (S&P’06)*, 2006, pp. 6 pp.–263.

[4] K. Karakaya and E. Bodden, "Sootfx: A static code feature extraction tool for java and android," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 181–186. [Online]. Available: https://doi.org/10.1109/SCAM52516.2021.00030

[5] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 54–61. [Online]. Available: https://doi.org/10.1145/379605.379665

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[7] L. Luo, E. Bodden, and J. Späth, "A qualitative analysis of android taint-analysis results," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 102–114.

[8] N. Heintze and O. Tardieu, "Demand-driven pointer analysis," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 24–34. [Online]. Available: https://doi.org/10.1145/378795.378802

[9] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proceedings of the 12th International Conference on Compiler Construction*, ser. CC'03. Berlin, Heidelberg: Springer-Verlag, 2003, p. 153–169.

[10] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[11] J. Späth, K. Ali, and E. Bodden, "Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems," vol. 3, no. POPL, jan 2019. [Online]. Available: https://doi.org/10.1145/3290361

[12] T. Reps, S. Schwoon, S. Jha, and D. Melski, "Weighted pushdown systems and their application to interprocedural dataflow analysis," *Science of Computer Programming*, vol. 58, no. 1, pp. 206–263, 2005, special Issue on the Static Analysis Symposium 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642305000493

[13] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 693–706. [Online]. Available: https://doi.org/10.1145/3192366.3192418

[14] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 265–266. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[15] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi, "Design and implementation of sparse global analyses for c-like languages," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 229–238. [Online]. Available: https://doi.org/10.1145/2254064.2254092

[16] Y. Sui, S. Ye, J. Xue, and P.-C. Yew, "Spas: Scalable path-sensitive pointer analysis on full-sparse ssa," in *Programming Languages and Systems*, H. Yang, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 155–171.

[17] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 2011, pp. 289–298.

[18] ——, "Semi-sparse flow-sensitive pointer analysis," *SIGPLAN Not.*, vol. 44, no. 1, p. 226–238, jan 2009. [Online]. Available: https://doi.org/10.1145/1594834.1480911

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

[20] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 49–61.

[21] R. Padhye and U. P. Khedker, "Interprocedural data flow analysis in soot using value contexts," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, ser. SOAP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 31–36. [Online]. Available: https://doi.org/10.1145/2487568.2487569

[22] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 264–280. [Online]. Available: https://doi.org/10.1145/353171.353189

[23] "secure-software-engineering/pointerbench: A points-to and alias analysis benchmark suite," https://github.com/secure-software-engineering/PointerBench, (Accessed on 10/04/2022).

[24] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 2, may 2020. [Online]. Available: https://doi.org/10.1145/3381915

[25] S. Ye, Y. Sui, and J. Xue, "Region-based selective flow-sensitive pointer analysis," in *Static Analysis*, M. Müller-Olm and H. Seidl, Eds. Cham: Springer International Publishing, 2014, pp. 319–336.

[26] T. Tan, Y. Li, and J. Xue, "Making k-object-sensitive pointer analysis more precise with still k-limiting," in *Static Analysis*, X. Rival, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 489–510.

[27] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 230–241. [Online]. Available: https://doi.org/10.1145/178243.178263

[28] J.-D. Choi, R. Cytron, and J. Ferrante, "Automatic construction of sparse data flow evaluation graphs," in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 55–66. [Online]. Available: https://doi.org/10.1145/99583.99594

[29] J. Lerch, B. Hermann, E. Bodden, and M. Mezini, "Flowtwist: Efficient context-sensitive inside-out taint analysis for large codebases," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 98–108. [Online]. Available: https://doi.org/10.1145/2635868.2635878

[30] J. Wang, Y. Wu, G. Zhou, Y. Yu, Z. Guo, and Y. Xiong, "Scaling static taint analysis to industrial soa applications: A case study at alibaba," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1477–1486. [Online]. Available: https://doi.org/10.1145/3368089.3417059

[31] X. Zhang, X. Wang, R. Slavin, and J. Niu, "Condysta: Context-aware dynamic supplement to static taint analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 796–812.

[32] N. A. Naeem and O. Lhotak, "Typestate-like analysis of multiple interacting objects," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, ser. OOPSLA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 347–366. [Online]. Available: https://doi.org/10.1145/1449764.1449792

[33] N. A. Naeem, O. Lhoták, and J. Rodriguez, "Practical extensions to the ifds algorithm," in *Compiler Construction*, R. Gupta, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 124–144.

[34] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, may 2008. [Online]. Available: https://doi.org/10.1145/1348250.1348255

[35] "Codeshield-security/spds: Efficient and precise pointer-tracking data-flow framework," https://github.com/CodeShield-Security/SPDS, (Accessed on 09/22/2022).

[36] "secure-software-engineering/flowdroid: Flowdroid static data flow tracker," https://github.com/secure-software-engineering/FlowDroid, (Accessed on 09/22/2022).

[37] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[38] "Free android market data, history, rankings — since 2011," https://www.androidrank.org/, (Accessed on 09/29/2022).

[39] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[40] F. Wilcoxon, *Individual comparisons by ranking methods*. Springer, 1992.