Shooting from the Heap: Ultra-Scalable Static Analysis with Heap Snapshots

Neville Grech
Dept. of Informatics, University of Athens
and Dept. of Computer Science, University of Malta
me@nevillegrech.com

George Fourtounis
Dept. of Informatics, University of Athens
Greece
gfour@di.uoa.gr

Adrian Francalanza
Dept. of Computer Science, University of Malta
Malta
adrian.francalanza@um.edu.mt

Yannis Smaragdakis
Dept. of Informatics, University of Athens
Greece
yannis@smaragd.org

ABSTRACT

Traditional whole-program static analysis (e.g., a points-to analysis that models the heap) encounters scalability problems for realistic applications. We propose a "featherweight" analysis that combines a dynamic snapshot of the heap with otherwise full static analysis of program behavior. The analysis is extremely scalable, offering speedups of well over 3x, with complexity empirically evaluated to grow linearly relative to the number of reachable methods. The analysis is also an excellent tradeoff of precision and recall (relative to different dynamic executions): while it can never fully capture all program behaviors (i.e., it cannot match the near-perfect recall of a full static analysis) it often approaches it closely while achieving much higher (3.5x) precision.

CCS CONCEPTS

• **Software and its engineering → Compilers**; *General programming languages*; **General programming languages**;

KEYWORDS

Program Analysis, Heap Snapshots, Scalability

ACM Reference Format:
Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2018. Shooting from the Heap: Ultra-Scalable Static Analysis with Heap Snapshots. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3213846.3213860

1 INTRODUCTION

Static analysis [27] attempts to infer the behavior of a program under all possible inputs. The technique usually strives to infer over-approximate behavioral abstractions such as call-graphs and points-to sets, by capturing the entire space of possible executions of the program. These abstractions can then be used to optimize compilation, assist program comprehension, and facilitate bug-finding, potentially revealing errors that only manifest themselves long after the analyzed software is released and deployed [2].

The main advantage of static analysis is, thus, *completeness*, *i.e.*, a perfect (or, in practice, near-perfect) *recall* of actual behaviors. But despite its effectiveness, static analysis generally suffers form two main weaknesses: (*i*) it is computationally expensive and suffers from *scalability* issues (*ii*) it can lack *precision*, typically producing a large amount of false positives, *i.e.*, predicted behaviors that do not match actual executions.

By contrast, dynamic analysis techniques such as testing [25], dynamic typing [20], and runtime monitoring and verification [12] avoid computationally expensive analyses by limiting themselves to the information exhibited by the executing program, also taking advantage of runtime information learned such as concrete parameter and memory values (which would otherwise be hard to infer statically). The main downside of such approaches is that, often, they are not exhaustive (*e.g.* in [13] theoretical maximality results are established that substantially limit the properties that can be analysed at runtime). Put differently, programs can have an exponential number of different executions and sampling behaviors from any finite numbers is unlikely to capture all executions.

The space of program analysis is, therefore, defined by competing quality criteria: *completeness*, *precision*, and *scalability*. This paper proposes a tradeoff that attempts to combine limited dynamic information inside a static analysis, in order to drastically improve scalability and precision, at some cost to completeness (*i.e.*, the recall of actual executions).

Although valuable combinations of runtime and static analyses have been explored extensively (*e.g.*, dynamic-symbolic execution [23, 33]) these emphasize a *local* generalization of concrete values, in an attempt to induce more dynamic behaviors. The results remain incomplete, with good coverage locally but with little effort to simulate all possible whole-program behaviors. For instance, a dynamic-symbolic exercising of a method `foo` does not explore in full generality methods in a deep call chain (*e.g.*, 10 nested calls away) from `foo` but instead considers their behavior from a single concrete execution. Similarly, combining dynamic information in an otherwise fully-general whole-program static analysis has been a fruitful avenue, with tools such as `Tamiflex` [4] and, recently, `HeapDL` [15]. Such work aims to enhance *completeness*, permitting

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213860

the static analysis to cover behaviors of hard-to-analyze code, such as reflection calls, but does not address the *scalability* or *precision* limitations of static analysis.

In contrast, our *featherweight hybrid analysis* approach entails replacing parts of the static analysis hindering scalable reasoning with dynamic facts. Specifically, we replace static heap reasoning with *concrete heap snapshots*, derived from actual executions. Technology-wise, we are explicitly inspired by the HeapDL work [15], which argues that heap snapshots are a mature, portable, and industrial-strength technology in modern runtimes, such as the JVM and the Dalvik VM (for Android). A heap snapshot indeed captures a wealth of information about the program's actual behavior, in addition to the standard links between heap objects, and can report near-complete call-graphs of past program activity.

Featherweight hybrid analysis leverages the information of heap snapshots to enhance analysis precision and scalability. The result is a static analysis that is kept lightweight by sacrificing some of its generalizing elements. Other aspects of the static analysis remain unchanged: all values propagate inter-procedurally through method calls, of any complexity and context depth. The result exhibits several desirable properties:

- Very high scalability (with at least 3x and often over 10x speedups, and scaling to deeper context sensitivity), thus making the approach feasible for many applications.
- High completeness, in terms of several analysis metrics—*e.g.*, an over 96% recall of call-graph edges observed in different dynamic executions. The level of completeness approaches that of a full-fledged static analysis, at a fraction of the cost, and vastly exceeds that of a purely-dynamic analysis.
- High precision (2.2x to 3.5x higher, for different metrics) when compared to state-of-the-art static analysis.

Our main contributions are as follows:

- We posit the idea that static analysis scalability is primarily hindered by the exhaustive modeling of the heap, which is a central, shared data structure, as opposed to method calling (*i.e.*, the stack), which induces more localized value sharing.
- We formulate a static analysis that does not model *mutation* of the heap, replacing this modeling with dynamic snapshots of heap state. Heap values can be read, but they only correspond to actually observed values.
- We evaluate the resulting featherweight hybrid analysis. We analyze experimentally the tradeoff between precision, scalability, and completeness, showing how a traditional static analysis pays a heavy price (of imprecision and cost) for completeness that may be unnecessary. The evaluation captures the potential from multiple angles: running time, asymptotic trends, precision/recall tradeoffs. Our purposely partial but lightweight analysis often approaches the completeness of much less scalable but semantically-complete static analyses.

2 BACKGROUND AND MOTIVATION

In this work, we focus on *whole-program* analysis, which models global structures, such as the *stack* and the *heap* in conventional programming languages. In particular, we are interested in *points-to analysis* [32, 35, 38], an analysis of how values flow through references throughout the program. Such a large-scale analysis cannot afford to model a program in full detail: points-to analyses are rarely *path-sensitive* (*i.e.,* modeling separately all possible execution paths, or distinct combinations of branches) or even *flow-sensitive* (modeling different branches separately). Instead, the analysis typically keeps a single global view of the entire heap, which represents the union of all possible sets of objects that a variable or field may reference at *any* point of *any* execution.

This simplifying view of program behavior allows points-to analysis algorithms to handle flow of information from extremely far-away program sites. In contrast to techniques that closely model program semantics (such as model checking, data-flow analyses, or variants of symbolic execution), points-to analysis accounts in full generality for the effects of distant or flow-unrelated methods (*e.g.*, methods that got called billions of instructions earlier during an actual execution, or methods that are hundreds of calls above or below the current method in any calling stack).

This ability comes at a cost: points-to analysis typically offers low precision and suffers from scalability issues. To control the balance between precision and scalability,[1] points-to analyses often employ *context-sensitivity*: instead of computing information that unifies all possible executions, the analysis only groups together different executions that have the same context. Each set of abstract values computed (*e.g.*, the possible values of a method's variables or of an object's fields) is qualified under a context. Context can take many shapes [35]. *E.g.*, a single-element context for the variables of a method can be the call-site of the method, or the object on which the method was called.

Striking a good balance between scalability and precision is far from a resolved question [21]. Typical current whole-program analyses easily become unscalable (see, *e.g.*, [37]) for a context of depth merely 2, which is not even precise enough to differentiate separate instances of common data structures such as a Java HashSet.

The main insight underpinning our hybrid analysis is that the handling of the global heap (as opposed to the much more disciplined stack structure) is the main culprit for the unscalability of static analysis. In a conventional language, like Java, the heap is a graph structure containing objects as nodes and references as edges, which may take multiple forms, *e.g.*, object field references, static field references or array content references. There are several reasons why the static modeling of the heap is unscalable. The heap is a global structure, therefore the effects of unrelated program parts accumulate and can affect any other program parts. The heap also includes static references—the analogue of global variables in the imperative world—which can typically be modified by any part of the program. Furthermore, array contents are typically collapsed to a single variable abstraction (due to lack of precise, path-sensitive reasoning), which results in large imprecision, and, hence, unscalability.

3 FEATHERWEIGHT HYBRID ANALYSIS

We next describe our hybrid analysis approach, consisting of a static analysis of the program, yet considering only a dynamic view of the program's heap. This combination aims to achieve scalability, far

[1] The precision vs. scalability tradeoff is complex. Scalability requires at least *good enough* precision—low precision will render an analysis unscalable unless special representations are employed. However, high precision will also make an analysis unscalable: the analysis model blows up in size, in a standard state-explosion fashion.

beyond typical static analyses, and coverage of different program behaviors, far beyond a purely dynamic analysis.

3.1 Featherweight Static Analysis

Since the modeling of the heap is the main source of unscalability for static analysis, is there a way to avoid modeling it? For most conventional languages, such an analysis would appear to be useless from a purely static stance since a static analysis that does not model the heap is grossly incomplete, missing most valid program behaviors. However, our intention is to combine concrete models of the heap with a static analysis that can *integrate* these models, yet does not otherwise itself compute models of the heap.

Our *featherweight static analysis* realizes this aim. It is a points-to analysis that fully models inter-procedural elements, such as the stack (*i.e.,* method calls, parameter passing, and method return values). As regards to the heap, the analysis can only *read in* information. Specifically, it only models heap loads (*e.g.*, reading from fields and arrays) but not stores (*e.g.*, field updates): the latter information must come from dynamic heap models.

We give a formal model of the featherweight analysis for a minimal intermediate language, shown in Figure 1. Other constructs, such as control flow, are irrelevant to the analysis (since it is flow-insensitive) and type information is modeled in separate relations, discussed later. The analysis is a standard Andersen-style points-to analysis with on-the-fly call-graph construction [35, 38]. The analysis also has parametric context-sensitivity, which we describe in more detail later. Programs are assumed to be sets of instructions with variables having a unique name per method (in general, per nested block scope). Instruction labels are assumed unique.

Input, Output, and Conventions. The domains of the analysis (and meta-variables used subsequently, plain or primed) comprise the following: a set of variables, $\mathtt{v}, \mathtt{u} \in V$, a set of methods, $\mathtt{meth} \in M$, a set of instruction labels, $i, j \in I$, a set of fields, $\mathtt{f} \in F$, a set of class types, $\mathtt{T} \in T$, a set of contexts, $c \in C$, a set of heap contexts, $h \in H$, and a set of abstract objects, $o^i_h \in O$. We use instruction labels, i, to annotate `new` instructions, method calls, and returns—implicitly, all instructions have a label, but it is not used in the model for other instruction types. An abstract object, o^i_h, uniquely identifies its allocation instruction (i.e., $\mathtt{v} = \mathtt{new}\ \mathtt{T}_i\ ()$), via its superscript i, and its allocation heap context, via its subscript h. We omit the superscript or subscript of an abstract object, writing just o, whenever they are not important (but merely propagate unchanged). In addition to a set of instructions, the analysis has access to standard symbol-table/type system information:

- We write $\mathtt{meth}_\mathtt{T}$ to denote the result of looking up (per the usual overriding rules) method signature `meth` in a class type `T`.
- We represent formal argument of method `meth` at position n with $arg_n^{\mathtt{meth}}$.
- For a method call instruction with label i, we represent its actual argument at position n as arg^i_n.
- We overload set membership notation: $o \in \mathtt{T}$ means that an abstract object o is of type `T`; $i \in \mathtt{meth}$ means that the instruction with label i is in the body of method `meth`.

The interfacing of the featherweight static analysis with its surroundings (including dynamic analysis inputs and final outputs) is done via the relations shown in Figure 2. For each relation, we note if it is an *input* that has already been filled in by the dynamic step, or a relation *computed* by the featherweight static analysis. Note that relations *Reachable* and *CallGraphEdge* are both input and computed: they initially contain the facts of the dynamic step and that information is then augmented by the featherweight analysis. Importantly, relation *FieldPointsTo* is input-only: any information on the shape of the heap comes from the outside world.

Static Analysis Inference Rules. The analysis model works by applying the rules in Figure 3 iteratively until fixpoint. Most rules in Figure 3 correspond to one of the instruction forms in Figure 1—for clarity, the instruction is listed as the first premise of such rules. The static analysis considers the respective instructions in any order: the rules are monotonic, so their order of application over many iterations reaching fixpoint does not influence the outcome.

The first rule to apply is necessarily Alloc, since all others require a *VarPointsTo* assertion, i.e., $\_ \longrightarrow_c o$, as one of their premises (which is in turn generated by this inference rule). Stated otherwise, the analysis begins from an initially reachable set of methods (provided as input) and infers points-to relationships, which trigger further inferences according to the program instructions.

Context-sensitivity is parametric, following a model introduced in reference [36]. This gives us the flexibility of choosing different flavors of context sensitivity. Concretely, the analysis is supplied with two helper functions:

- $\mathcal{NHC}(i, c)$ (also known as Record [36]) is used in rule Alloc and creates a *new heap context* for instruction i and context c,
- $\mathcal{NC}(i, c, o)$ (also known as Merge [36]) is used in rule Call) and constructs a *new (callee) context* for instruction i, (caller) context c, and abstract object o.

The definition of these functions is purposely left abstract and can be instantiated to get the desired flavor of context sensitivity.

As we discussed earlier, the analysis model covers load instructions but *not* store instructions. As a result, when this analysis is exercised in stand-alone fashion, it does not infer heap state, *i.e., FieldPointsTo* assertions, which in principle prevents rules such as Load from being applied. However, our hybrid analysis assumes that the dynamic analysis will generate the necessary *FieldPointsTo*, which are then used by other inference rules such as Load.

The Call rule infers two types of assertions: it establishes call-graph edges (based on points-to information) and a reachability assertion for a newly created context c' (using standard dynamic lookup $\mathtt{meth}_\mathtt{T}$) of a method signature). The rule is separate from the inferences made by the Args rule, since failure to infer points-to values for an argument should not prevent the establishment of a call-graph edge and target-method reachability. The Args and Ret rules cover propagation of points-to information on the stack, via method parameters and returns.

Notably, in this minimal language, the heap only consists of instance field references (*i.e.,* no static fields or arrays). Hence, the featherweight static analysis is only missing a rule to handle store instructions, compared to a fully static analysis. Such a rule would have the form:

$$(\textsc{Store})\quad \frac{\mathtt{u.f = v} \qquad \mathtt{u} \longrightarrow_c o \qquad \mathtt{v} \longrightarrow_c o'}{o.\mathtt{f} \longrightarrow o'}$$

$$
\begin{array}{lcll}
\mathit{Instruction} & := & \mathtt{v = new\ T}_i\,() & \mathit{new} \\
 & | & \mathtt{v = u} & \mathit{move} \\
 & | & \mathtt{u = v.f} & \mathit{field\ load} \\
 & | & \mathtt{u.f = v} & \mathit{field\ store} \\
 & | & \mathtt{u} =_i \mathtt{v.meth(v_1, \ldots, v}_k\mathtt{)}, \quad k \geq 0 & \mathit{method\ call} \\
 & | & \mathtt{return}_i\ \mathtt{v} & \mathit{return}
\end{array}
$$

Figure 1: Syntax of the source language instructions.

Relation	Notation	Description	Input	Computed
FieldPointsTo	$o_h.\mathtt{f} \rightarrow o'_{h'}$	Field f of abstract object o_h points to abstract object $o'_{h'}$.	✓	
VarPointsTo	$\mathtt{v} \rightarrow_c o^i_h$	Variable v, in context c, points to abstract object o^i_h with heap context h via allocation instruction i.		✓
Reachable	$\overline{\mathtt{meth}}^c$	Method meth is reachable in context c.	✓	✓
CallGraphEdge	$i \overset{c}{\xrightarrow{calls}}_{c'} \mathtt{meth}$	Instruction i calls method meth, under caller context c and callee context c'.	✓	✓

Figure 2: Relations used in the featherweight static analysis.

$$
(\textsc{Alloc})\ \frac{\mathtt{v = new\ T}_i\,() \qquad i \in \mathtt{meth} \qquad \overline{\mathtt{meth}}^c \qquad h = \mathcal{NHC}(i,c)}{\mathtt{v} \rightarrow_c o^i_h}
\qquad
(\textsc{Load})\ \frac{\mathtt{u = v.f} \qquad \mathtt{v} \rightarrow_c o \qquad o.\mathtt{f} \rightarrow o'}{\mathtt{u} \rightarrow_c o'}
$$

$$
(\textsc{Move})\ \frac{\mathtt{v = u} \qquad \mathtt{u} \rightarrow_c o}{\mathtt{v} \rightarrow_c o}
\qquad
(\textsc{Call})\ \frac{\mathtt{u} =_i \mathtt{v.meth(v_1, \ldots, v}_k\mathtt{)} \qquad \mathtt{v} \rightarrow_c o \qquad o \in \mathtt{T} \qquad c' = \mathcal{NC}(i,c,o)}{\overline{\mathtt{meth}_\mathtt{T}}^{c'} \qquad i \overset{c}{\xrightarrow{calls}}_{c'} \mathtt{meth}_\mathtt{T}}
$$

$$
(\textsc{Args})\ \frac{i \overset{c}{\xrightarrow{calls}}_{c'} \mathtt{meth} \qquad arg^i_n \rightarrow_c o}{arg^{\mathtt{meth}}_n \rightarrow_c o}
\qquad
(\textsc{Ret})\ \frac{\mathtt{return}_i\ \mathtt{v} \qquad i \in \mathtt{meth} \qquad \mathtt{v} \rightarrow_c o \qquad j \overset{c}{\xrightarrow{calls}}_{c'} \mathtt{meth} \qquad \mathtt{u} =_j *}{\mathtt{u} \rightarrow_c o}
$$

Figure 3: Inference rules for featherweight static analysis.

3.2 Dynamic Analysis

To produce our hybrid analysis, featherweight static analysis needs to be supplied with dynamic heap information from standard Java HPROF heap dumps, which are snapshots of the heap and stack during program execution [28].

Following Grech et al. [15], our HPROF heap dump analyzer leverages the standard cross-platform support for heap profiling in Java-based runtime environments, *i.e.,* the ability to produce whole-heap snapshots containing all objects allocated in the heap, together with references between these objects (the shape of the heap). Heap snapshots are analyzed offline. Heap snapshots also offer insights about the stack shape: they include full stack traces produced when any object is created. Grech et al. [15] summarize this capability of modern heap snapshot technology as "*a typical heap snapshot also integrates many thousands of stack snapshots*".

Typically, heap dumps reflect a substantial portion of the complex dynamic behavior of a program, regardless of the cause of such behavior: instead of watching what happens at specific actions (*e.g.,* reflection or dynamic loading operations), a heap dump records the cumulative semantic effects of program execution in its native setting and complex environment. At the same time, heap dumps do not miss the ability to capture dynamic actions (*e.g.,* a dynamic call-graph) since each object records information describing the dynamic context at the time of its allocation.

The heap dump analysis works by iterating over the heap snapshot to build a graph structure of the heap. In the process it resolves forward and backwards references, indexes instances by class hierarchy information, and more. Object abstractions are mainly derived from allocation traces (*i.e.,* call stacks), which are traversed until the actual new object expression is found in the application. This is not always straightforward, since some code may occasionally be missing. In this case heuristics are used to predict the stack frame where the actual object was allocated. If allocation traces are not present in the heap, a coarser-grain abstraction is currently constructed per-type in case of application classes. Otherwise a finer grain abstraction is used according to the object's contents. Given these tools to traverse the heap snapshot and form suitable object abstractions, the following dynamic heap information is extracted:

Object Field Values: values an object's fields can point to, for the entire class hierarchy of the object's class.

Static Field Values: values a class's static fields can point to.

Array Content Values: values an array's contents can point to.

In addition, dynamic control-flow information is also extracted from heap snapshots. Each allocation trace present in the snapshot represents a path through the *call graph.* These paths are combined to form a call graph, with every successive node in the trace being an edge in this call graph. Similarly, the allocation trace is a witness that every method involved in the trace is *reachable*.

To augment the information that is normally found in a heap snapshot, we force the program to store more objects on the heap. This is usually achieved via program instrumentation: specially designed Java agents perform load-time structured bytecode transformations so that new objects and references to existing objects are created at strategic program sites, and thus reflected in the snapshot. This allows additional information to be extracted from the enriched heap. Such information includes: (*i*) Various kinds of *context sensitivity* information, used by call-site- and object-sensitive analyses [26, 34]. This is done by storing additional information at program points where objects are allocated (*e.g.,* the current receiver object—a concrete heap context—is used for object sensitivity); (*ii*) Information about *dynamically loaded code* linked to its class and class loader; (*iii*) *Dead objects* that reflect some past run-time behavior. By creating these additional references, many extra objects are live when the snapshot is taken.

Although heap dumps and dynamic stack traces capture a wealth of information, an analysis based solely on them does not attain adequate levels of method coverage (*i.e.*, lacks in completeness, measured in terms of *recall*) when compared to (standard) static analyses. For instance, we computed that this dynamic analysis only covers an average of 12% of application methods in the Dacapo 2009 benchmark suits, even though the benchmark inputs are thorough. For instance, the jython benchmark executes pyBench, which is meant to exercise most features of the Python language. In spite of this, only around 10% of the application's methods are covered. We provide a more detailed discussion of our evaluation in Section 5.

3.3 Combining the Analyses

Combining the featherweight static analysis with dynamic snapshots of the heap should yield several benefits. The hybrid analysis can exploit the strengths of the dynamic analysis while avoiding some of the weaknesses of whole-program Java pointer analyses; and vice versa. Compared to a full-blown static analysis, the main benefit of featherweight analysis is scalability: the dynamic analysis excels in its ability to get very precise heap information, which still fully captures the observed dynamic behaviors (but possibly not others). Compared to a stand-alone dynamic analysis, the expected benefit is higher completeness in the coverage of possible program behaviors. Although the heap image is fixed, it is possibly rich enough to allow thorough static exploration of program state. The approach of being complete only for the observed dynamic execution, yet opportunistically generalizing, is reminiscent of other techniques, such as *dynamic-symbolic execution* [14]. However, the featherweight hybrid analysis has stack variables analyzed in full static generality, in a whole-program fashion. Such propagation of values can be far-reaching: *e.g.*, it can over-approximate the effects of unboundedly long sequences of method calls.

It is important to stress that a featherweight hybrid analysis *is* a static analysis. It emphasizes the over-approximate nature of static abstractions. *E.g.*, when multiple callers of a method pass different values as a parameter, the values are collected as a set and (whenever the different callers are not distinguished based on the current context-sensitivity policy) their results are considered collectively. Thus, each caller can see results that pertain to all others. This is an important distinction between an approach that is fundamentally static and an approach (such as dynamic-symbolic execution) that is fundamentally dynamic: when a static analysis encounters difficulty in modeling a feature, it prefers to be imprecise but complete. When a dynamic analysis encounters the same difficulty, it opts for precision instead of an over-approximation.

4 IMPLEMENTATION

We implemented our featherweight hybrid analysis over the Doop [5] pointer and taint [18] analysis framework. Doop is full-featured and handles several complex semantic aspects of the Java language, such as reflection, implicit initialization, exceptions, and more. For dynamic analysis we used HeapDL [15], which processes HPROF heap dumps produced by Java's heap profiler.

The featherweight hybrid approach applies orthogonally to all analyses in the Doop framework. This is important, since Doop is distinguished by its richness in supporting different kinds of context-sensitivity. Some-40 different context-sensitive analyses are in the current main Doop code, with tens more labeled "experimental" in the code base.

The full implementation touches significantly more elements than the formal model of Section 3. Heap information in the full analysis includes array contents, static fields and instance fields. In total, these elements characterize the points of mutation of complex objects in Java. Therefore, our implementation disables all analysis inferences that are based on heap *stores*, of any kind. This requires interventions to far-reaching logic, such as reflective actions, handling of arrays, native methods that are semantically equivalent to heap stores (*e.g.*, compareAndSwapObject), and more.

Additionally, we employ novel special-purpose refinements to the logic of [15] to integrate heap snapshots into a static analysis:

Singletons. A good example is the handling of singleton objects. If a heap snapshot reveals that a certain type only has a *single* instance (as a heap abstraction) ever created, then that instance is considered to be the sole representative of its type. The analysis then considers this instance to flow wherever there is a compatible receiver variable (this) for some call-graph edge that is confirmed dynamically, regardless of other static analysis inferences. This is an excellent heuristic, very much in the spirit of the featherweight hybrid analysis: it is inexpensive (since it only propagates a single object), does not sacrifice precision, but enhances completeness. Throughout our experimentation, we regularly find that more than 10% of object abstractions in heap dumps are singletons, making the heuristic widely applicable.

Missing Field Data. We also do limited static modeling of some incomplete heap object structures (*e.g.*, with transient fields) but only when these inferences are corroborated by other dynamic information. The philosophy of this heuristic is in line with featherweight analysis: we always prefer dynamic modeling of the heap, except where this dynamic information is clearly incomplete.

Tuning Reflection Analysis. The reflection-handling logic in the featherweight analysis is tuned accordingly. The Doop framework's reflection logic is highly tunable, with several completeness vs. precision knobs. Since featherweight analysis achieves much higher precision than a full static analysis, we disable or turn down aggressive heuristics (*e.g.*, we disregard short string constants in reflection inferences) aimed at countering static analysis imprecision.

5 EXPERIMENTAL EVALUATION

We next present the results of an experimental evaluation of the featherweight analysis. We evaluate using the DaCapo 9.12-Bach Java benchmark suite [3]. (Some benchmarks are excluded *a priori* due to engineering reasons—*e.g.*, cannot run under a profiler—as also documented in past work [4, 15].) The results described are entirely representative of our experience with several other programs. However, we here report the DaCapo benchmarks for the evaluation because they offer a standardized, oft-used suite, of representative medium-sized Java applications, and also give us the opportunity to run the applications using different, pre-decided inputs so that we can validate static analysis results with multiple dynamic runs. This gives us an ability to evaluate completeness and cover unseen executions by the analysis.

We use the following analyses/configurations:

Dynamic-small: A dynamic analysis on the DaCapo benchmarks running a *small* input set. The output of this dynamic analysis forms a *baseline* and can be seen as *a training input*: the static analyses are supplied heap snapshots from the dynamic-small run.

Dynamic: Combined results from dynamic analyses on DaCapo benchmarks running the default and large input sets. Dynamic and dynamic-small naturally have a sizeable intersection.

Full-static: A Doop static pointer analysis (under the most common Doop setting: selectively context-sensitive) with reflection support, as well as a heap snapshot as input, *i.e.*, under maximum-completeness settings.

Featherweight: Our featherweight static pointer analysis (same context-sensitivity and other settings as full-static).

The evaluation intends to answer the following research questions:

RQ.A What is the relative burden and scalability of the featherweight hybrid analysis compared to the full-static analysis?

RQ.B What is the analysis completeness and precision (*i.e.*, (usual *recall* and *precision* metrics relative to all dynamic executions) of the featherweight analysis compared to a full static analysis?

To answer these questions, we employ the following metrics:

Reachable methods: the number of methods that are deemed to be reachable. We use this metric to compare the completeness and precision of the respective analyses.

Call-graph edges: the number of edges in the call graph.

Relevant static analysis time: the time required to run our analyses, inclusive of pre- and post-processing.

Total var points-to: the size of the *VarPointsTo* relation, the largest and most expensive output relation of our analyses. This metric correlates with relevant static analysis time and, even more so, with the cost of further client analyses that one may want to run.

Both static analyses use the Soufflé compiler, which compiles Datalog specifications onto C++ and into binaries. Heap snapshots are processed using HeapDL [15]. Notably, as reported in that work, collecting heap snapshots incurs a large overhead on the JVM: roughly a 20x slowdown of the application-under-analysis. This is largely due to engineering reasons in the reference HPROF implementation (*e.g.*, legacy implementation of the tool forces the garbage collector to run in single threaded fashion, which causes a bottleneck in today's multicore architectures); these overheads disappear on Android, and modern profilers advertise much faster execution. (The Android setup is one in which we commonly and fruitfully employ featherweight analysis.) Regardless, the heap snapshot is only collected once per application and an arbitrarily expensive static analysis can be subsequently performed with it.

Unless explicitly specified otherwise, both full-static and featherweight-hybrid analysis are run using a 1-call-site sensitive analysis that is applied to the subset of the methods that have values flow from their inputs to their outputs. All applications are analyzed with the Oracle JRE 8 libraries (rev 131). Both analyses are run with full-featured static handling of reflection. All runtimes are established on an idle machine with an Intel Xeon E5-2687W v4 3.00GHz with up to 512 GB of RAM. All experiments have a cutoff time of four hours (14400 seconds).

5.1 RQ.A: Scalability

The main motivation for featherweight analysis is *scalability*. Therefore we consider the relative runtime burden between the full-static and featherweight-hybrid analysis (RQ.A). The total analyses times (in seconds) are shown in Figure 4. Notably, we see that *for all benchmarks the featherweight analysis is at least 3.5x faster than full-static, and up to well over 10x faster, for larger applications.* For smaller applications, the featherweight analysis has constant overheads that dominate the runtime, such as precomputing properties for possibly unreachable code.

Furthermore, the featherweight analysis can be empirically seen to scale linearly, relative to the size of the analyzed code base. We show this using as a metric the growth of the size of the largest analysis relation, *VarPointsTo*. The size of this relation correlates with running time but is a truer indication of the burden of both the main analysis and followup uses of analysis results. For instance, a more efficient machine or Datalog engine may make the analysis run faster but it is rarely the case that the underlying algorithms of the analysis scale differently. Figure 5 indicates a roughly linear relationship between the size of the *VarPointsTo* relation and the running total number of reachable methods (including in library code) for our featherweight-hybrid analysis. In fact, a generalized linear model with a linear basis function attains a good goodness-of-fit value ($R^2 = 0.83$). On the other hand, the growth curve for the full-static analysis in the same figure strongly suggests super-linear behavior. In this case, the model with quadratic basis functions has a better fit ($R^2 = 0.97$). This is yet another indication that, irrespective or analysis timings, the scalability of the featherweight-hybrid analysis is inherently better.

Deeper context-sensitivity. The figures show a (selectively) 1-call-site-sensitive configuration; this allowed us to make broad comparisons with the full-static analysis, which we could scale (however

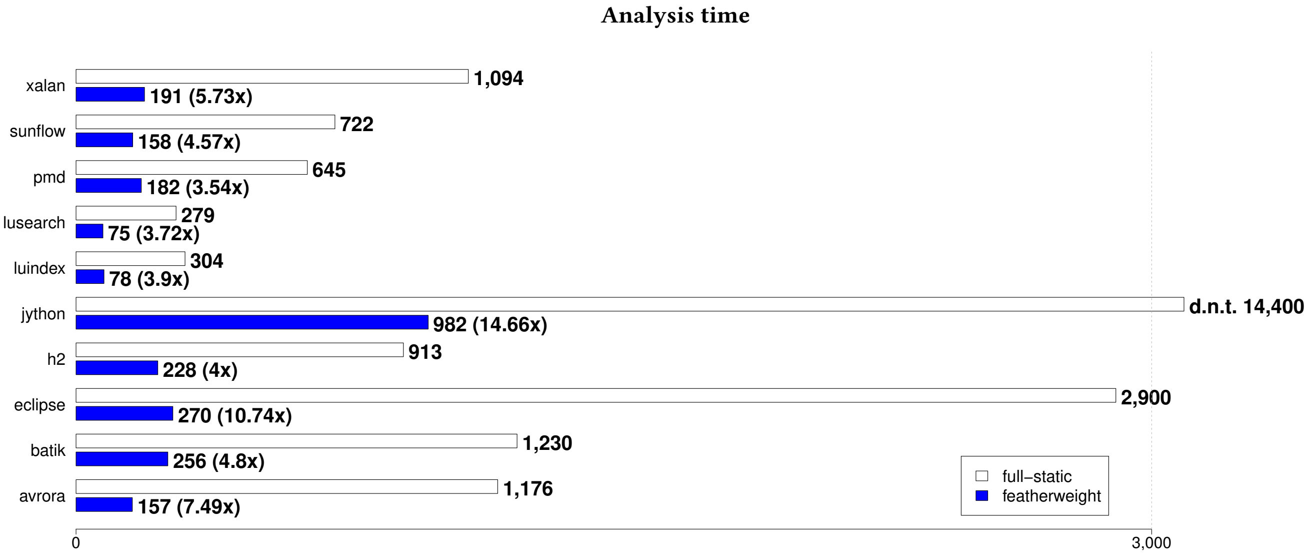

Figure 4: Analysis times in seconds and speed-up factors. The `jython` benchmark did not terminate in the given time under the full-static analysis, hence the speedup is a lower bound.

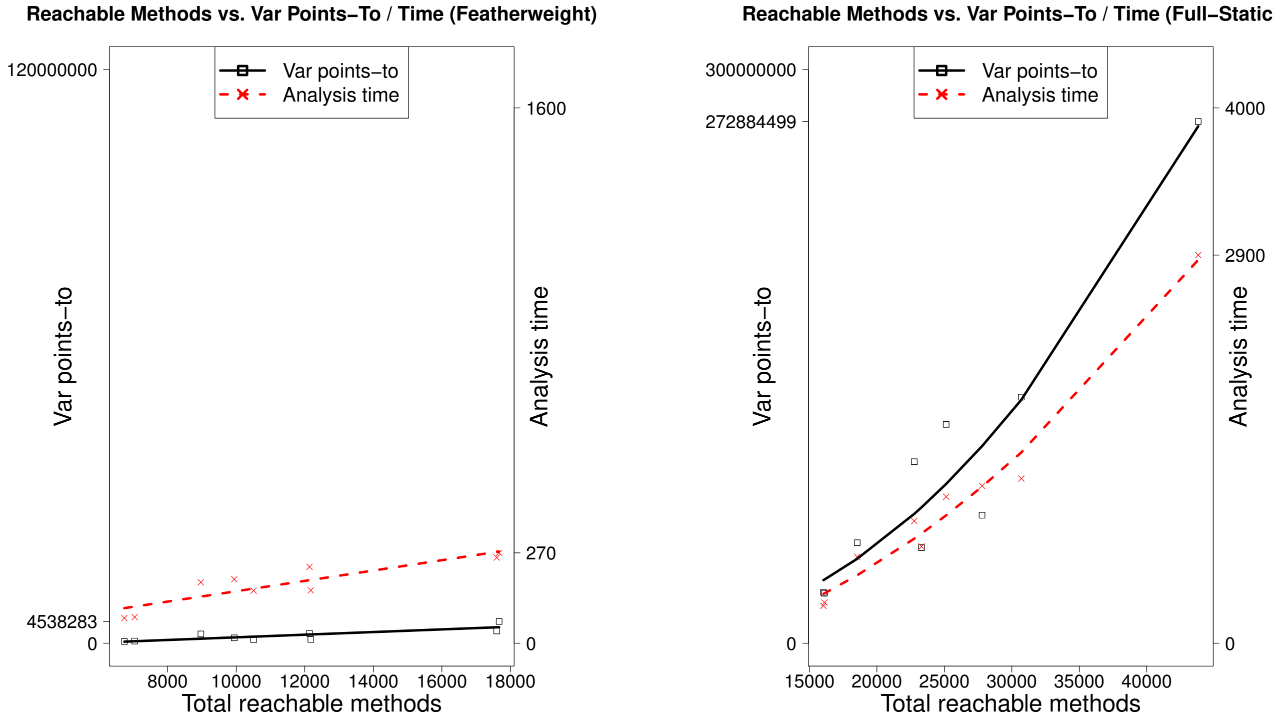

Figure 5: Relationship between time/var-points-to size and reachable methods for the featherweight (left) and the full-static (right) analyses. A linear and polynomial function fit the scatter plots of the featherweight and full-static analyses respectively. The two plots do not have the same maxima but they are scaled to similar aspect ratios for easier visual comparison of slopes (not absolute values).

imperfectly) under this flavor of context sensitivity. Nevertheless, the featherweight-hybrid analysis scales well even for *much deeper context sensitivity*. For instance, the analysis takes under 30 minutes for all benchmarks but `jython` for a selective-2objH analysis: a very precise form of context sensitivity that combines 2-object-sensitive and 1-call-site calling contexts and has a 1-object-sensitive heap context [22]. By contrast, this setup renders the full-static analysis completely unscalable—only the smallest benchmarks (`luindex` and `lusearch`) terminate in under 4 hours, with the featherweight analysis achieving a 36x and 41x speedup, respectively.

Table 1: Precision and Recall for Reachable Methods. The numbers are truncated instead of rounded, so that 100% occurs only for identical reports.

	precision		recall (all methods)		recall (unseen methods)	
	full-static	featherweight	full-static	featherweight	full-static	featherweight
avrora	9%	24%	100%	99%	100%	67%
batik	16%	28%	99%	96%	93%	68%
eclipse	14%	30%	99%	89%	94%	51%
h2	12%	23%	100%	100%	100%	92%
luindex	12%	26%	99%	96%	99%	70%
lusearch	9%	22%	99%	99%	100%	98%
pmd	15%	30%	100%	95%	99%	74%
sunflow	13%	24%	100%	100%	100%	100%
xalan	9%	23%	100%	100%	100%	100%
average	**12%**	**26%**	**99%**	**97%**	**98%**	**80%**

Table 2: Precision and Recall for Call Graph Edges. The numbers are truncated instead of rounded, so that 100% occurs only for identical reports.

	precision		recall (all edges)		recall (unseen edges)	
	full-static	featherweight	full-static	featherweight	full-static	featherweight
avrora	3%	13%	99%	95%	100%	72%
batik	7%	18%	98%	93%	92%	59%
eclipse	3%	13%	98%	88%	93%	49%
h2	3%	9%	100%	99%	100%	91%
luindex	4%	14%	99%	92%	98%	52%
lusearch	3%	11%	99%	99%	98%	90%
pmd	6%	19%	99%	94%	97%	81%
sunflow	4%	11%	99%	99%	100%	100%
xalan	2%	10%	99%	99%	100%	100%
average	**4%**	**14%**	**99%**	**96%**	**97%**	**77%**

5.2 RQ.B: Completeness and Precision

Featherweight analysis is scalable, but is it an *effective* static analysis? More precisely, we still need to show that it can predict, with good precision, behaviors that it has *not* seen as part of its input heap snapshot. To assess this, we compare the precision and recall (for reachable methods and call-graph edges) of the featherweight analysis and the full-static analysis. Ground truth for these measurements is taken to be the full set of available executions for the benchmarks (*i.e.*, not just the "small" benchmark input, used to produce the heap snapshot, but also "default" and "large" inputs).

Precision and Recall are tabulated in Tables 1 and 2. To focus on *unseen* executions, we also give recall results (under the heading "recall (unseen methods)") for methods that appear in the "default" and "large" run but not in the "small" run. As can be seen in the tables, the full-static analysis is highly complete, as would be expected of a state-of-the-art static analysis (especially one enhanced with reflection analysis logic *and* heap snapshots). For instance, the analysis captures over 99% (on average) of all methods and call-graph edges encountered under any dynamic input. Even if we narrow the denominator to the methods or call-graph edges *unseen* in the heap snapshot used as input, the full-static analysis still achieves over 97% or 98% recall.

The featherweight analysis is also good in terms of recall, however. It captures 97% (resp. 96%) of all methods (resp. call-graph edges) seen in dynamic executions. If narrowed to the methods not seen in the input heap snapshot, recall is still over 80% in terms of methods and over 77% in terms of call-graph edges.

Furthermore, the featherweight analysis offers a better "bang for the buck" and uses the computational resources more effectively when compared to the full-static analysis, representing a good tradeoff between precision and completeness. Its precision metrics are significantly higher (2.2x better for methods and 3.5x better for edges). More concretely, when the featherweight analysis predicts that a certain method/edge is reachable/realizable, this is much more likely to be the case than in the full-static analysis. This observation goes hand-in-hand with the analysis scalability issue: the featherweight analysis analyzes fully statically the stack (*i.e.*, values passed through arguments/returns of calls) which is a more precise data structure than the globally-shared heap.

The Venn diagrams of Figure 6 show in full detail the call-graph edges discovered by each of the analyses. (A very similar figure for reachable methods is omitted for space reasons.) The diagrams help complete the picture from Tables 1 and 2. Featherweight analysis finds the majority of the edges in dynamic executions (as seen in the earlier recall numbers), including the majority of the edges not seen in the input snapshot, while predicting proportionally many fewer edges than the full-static analysis (as seen in the earlier precision numbers). The Venn diagram numbers illustrate in detail how the featherweight analysis predicts 3.5x fewer call-graph edges, yet still captures over 77% of previously unseen edges.

Call Graph Edges

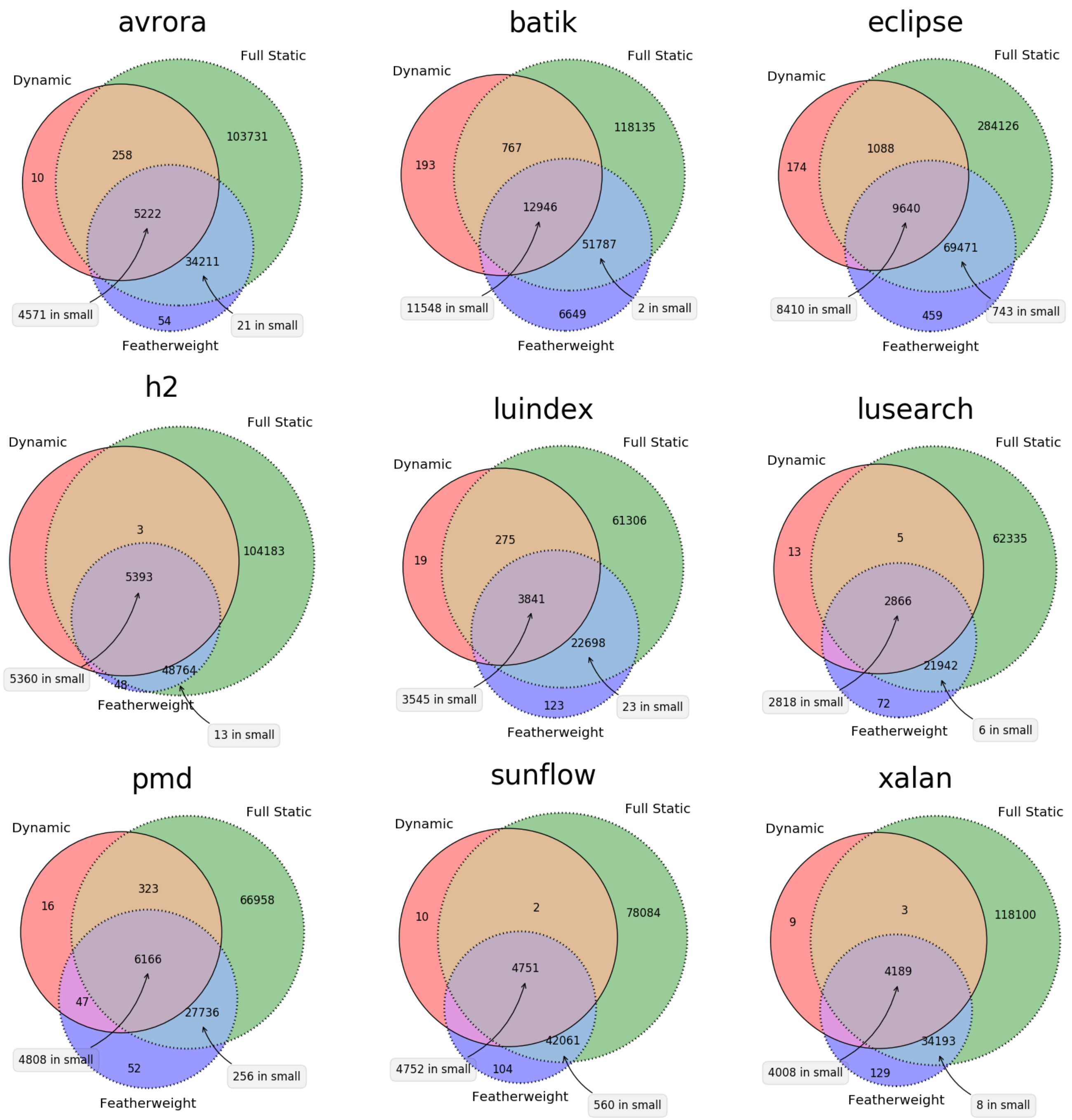

Figure 6: Venn diagrams depicting call graph sizes, discovered by each analysis configuration. For readability, we only show a 3-set Venn diagram, leaving the dynamic-small execution implicit. However, the information for dynamic-small is shown in the diagrams, using callouts. The dynamic-small results are (modulo very minor exceptions) a subset of the intersection of full-static and featherweight, since the heap snapshot from dynamic-small is supplied as an input to both static analyses.

6 RELATED WORK

The space of combinations of static and dynamic analyses is huge. Our discussion of related work is merely a sampling.

Using static analysis to optimize/reduce runtime checks. Using static analyses as part of a scheme for runtime verification is relatively common [6, 10, 17, 19, 31]. Our approach however first runs a dynamic step as input to a static analysis. The opposite, running first a static analysis to infer general properties that will be inputs to a dynamic analysis, is more common. There are many examples of this approach. For instance, Elkarablieh *et al.* [9] run a static analysis to guide dynamic data structure repair. Most forms of optimization (*e.g.* reducing runtime checks) require over-approximate static information. Our approach can be most closely compared with other work that uses dynamic information as input to static analyses, such as the combined dynamic information with static shape analysis of Raman and August [30] and Aftandilian *et al.* [1].

Heap snapshots. Potanin *et al.* run queries on heap snapshots to find bugs on a particular program run [29]. Our approach uses static analysis to generalize a heap snapshot to infer facts about other possible executions and thus can also generalize their technique to be applicable in a static context and find more bugs than those found in the original program. Flanagan and Freund abstract heap snapshots to construct UML-style object models [11]. Their abstract interpretation constructs abstract objects by only merging dynamic allocations while our pointer analysis uses both the heap-allocated objects observed at runtime and the statically allocated objects (via `new` in the source code).

Static-dynamic analysis combinations. A hybrid static-dynamic approach is that of combined symbolic-concrete ("concolic" or "dynamic-symbolic") execution of code [14, 33], usually employed for testing purposes. Concolic execution switches on the fly between an abstract analysis state and a runtime-values state, in an effort to satisfy symbolic conditions in the program text and cover its behavior thoroughly. These techniques have been refined in the Pex tool [40] to perform test case reduction and prioritization. The main benefit, as in our technique, is that concrete information can replace reasoning that would be too hard or impossible to perform statically. In this light, our approach is fully concrete for the heap and inferring the union of concrete and symbolic facts for the stack, without employing symbolic conditions. Another difference with concolic techniques lies in the type of the concrete information: ours is heap snapshots, each faithful to a whole-program run, instead of random tests that may exhibit more unexpected behavior.

Csallner *et al.* [7, 8] employ a hybrid analysis for bug finding, using a dynamic analysis to obtain higher confidence for error warnings compared to a plain static analysis. The same theme has been explored several times in the literature (*e.g.*, [24]). Taghdiri [39] uses a counterexample-guided refinement process to infer over-approximate specifications for procedures called in the function being verified. These approaches are different instances of combining dynamic and static information compared to our work, which focuses on addressing the scalability shortcomings instead.

Analysis for dynamic language features. Our work is closely related to `HeapDL` [15]. `HeapDL` is a cross-platform tool meant to supplement static analysis with dynamically-inferred information to improve the analysis *completeness*, especially for programs with highly dynamic features. `HeapDL` makes no direct contribution to analysis *scalability*. In contrast, our work explores the idea of eliminating parts of the static reasoning, expressly for scalability.

Similarly, the `Tamiflex` tool [4] captures run-time information about reflection. It works by instrumenting an application using Java agents, which intercept class loading and instrument bytecode at load time. The added instrumentation logs the parameters and program points where reflective operations are being used. `Tamiflex` only captures reflection information (which is important for static analysers with limited reflection support) but does not capture any other dynamic behavior. Thus, `Tamiflex` enhances static analysis completeness but does not supplant key parts of static reasoning, such as the analysis of the heap. In addition, recent improvements [16] in reflection analysis scalability have reduced the need to replace static reflection analysis with dynamic techniques.

7 CONCLUSIONS

We presented a featherweight hybrid analysis operating over a fixed model of the heap, established by dynamic analysis. The dynamic analysis leverages heap snapshots, produced using mainstream tools (standard-issue Java profilers) and a portable format (`HPROF`). The static analysis generalizes this information in terms of its impact on further stack-based behavior.

The combination yields benefits that are larger than the sum of the parts. The analysis is significantly more scalable than a full static analysis, especially under the most strenuous conditions (such as full static handling of reflection or deep context sensitivity). At the same time, the analysis has much better precision than a fully-static analysis. Overall, featherweight analysis has a higher completeness than a purely dynamic analysis, exposing a significantly higher number of reachable methods and even more call-graph edges.

Combinations of this flavor represent a fruitful direction to employ for analyses that have, so far, been entirely static, emphasizing generality and whole-program reach. Accordingly, there are several promising directions for future work. For instance, dynamic snapshots can be potentially combined with *flow-sensitive* analyses, by keeping per-instruction points-to information, possibly for entire program expressions (i.e., *access paths*). Furthermore, we are currently developing bug-finding analyses (mostly information-flow patterns) for large enterprise applications, in a joint project with a large industrial partner. Featherweight analysis is an excellent choice for such clients that (a) pose scalability challenges, (b) can tolerate incompleteness.

ACKNOWLEDGMENTS

We gratefully acknowledge funding by the European Research Council, grant 307334 (SPADE), a Facebook Research and Academic Relations award and an Oracle Labs collaborative research grant. In addition, the research work disclosed is partially funded by the REACH HIGH Scholars Program – Post-Doctoral Grants. The grant is part-financed by the European Union, Operational Program II, Cohesion Policy 2014-2020 (Investing in human capital to create more opportunities and promote the wellbeing of society - European Social Fund).

REFERENCES

[1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS '10)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/1879211.1879222

[2] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/1985793.1985827

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[6] Qichang Chen, Liqiang Wang, Zijiang Yang, and Scott D. Stoller. 2009. HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (FASE '09)*. Springer-Verlag, Berlin, Heidelberg, 425–439. https://doi.org/10.1007/978-3-642-00593-0_30

[7] Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 422–431.

[8] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. 2008. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. *ACM Transactions on Software Engineering and Methodology* 17, 2, Article 8 (May 2008), 37 pages. https://doi.org/10.1145/1348250.1348254

[9] Bassem Elkarablieh, Sarfraz Khurshid, Duy Vu, and Kathryn S. McKinley. 2007. STARC: Static Analysis for Efficient Repair of Complex Data. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 387–404. https://doi.org/10.1145/1297027.1297056

[10] Michael D. Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*. Portland, OR, 24–27.

[11] Cormac Flanagan and Stephen N. Freund. 2006. Dynamic Architecture Extraction. In *Proceedings of the First Combined International Conference on Formal Approaches to Software Testing and Runtime Verification (FATES'06/RV'06)*. Springer-Verlag, Berlin, Heidelberg, 209–224. https://doi.org/10.1007/11940197_14

[12] Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. 2017. A Foundation for Runtime Monitoring. In *Runtime Verification (RV) (LNCS)*, Vol. 10548. Springer, 8–29. https://doi.org/10.1007/978-3-319-67531-2_2

[13] Adrian Francalanza, Luca Aceto, and Anna Ingolfsdottir. 2017. Monitorability for the Hennessy–Milner logic with recursion. *Formal Methods in System Design* (2017), 1–30.

[14] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[15] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2017. Heaps Don't Lie: Countering Unsoundness with Heap Snapshots. *Proc. ACM Programming Languages (PACMPL)* OOPSLA (Oct. 2017), 68:1–68:27. Issue 1. https://doi.org/10.1145/3133892

[16] Neville Grech, George Kastrinis, and Yannis Smaragdakis. 2018. Efficient Reflection String Analysis via Graph Coloring. In *Proceedings of the 32nd European Conference on Object-Oriented Programming (ECOOP'18)*, Vol. 109. LIPICS, Leibniz, Germany, Article 26, 25 pages.

[17] Neville Grech, Julian Rathke, and Bernd Fischer. 2013. Preemptive type checking in dynamically typed languages. In *International Colloquium on Theoretical Aspects of Computing*. Springer, Springer-Verlag, Berlin, Heidelberg, 195–212. https://doi.org/10.1007/978-3-642-39718-9_12

[18] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-to and Taint Analysis. *Proc. ACM Programming Languages (PACMPL)* 1, OOPSLA, Article 102 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133926

[19] Rajiv Gupta, Mary Lou Soffa, and John Howard. 1997. Hybrid Slicing: Integrating Dynamic Information with Static Analysis. *ACM Transactions on Software Engineering and Methodology* 6, 4 (Oct. 1997), 370–397. https://doi.org/10.1145/261640.261644

[20] Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press, New York, NY, USA.

[21] Michael Hind. 2001. Pointer analysis: haven't we solved this problem yet?. In *Proc. of the 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*. ACM, New York, NY, USA, 54–61.

[22] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-To Analysis. In *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA.

[23] B Korel, H Wedde, and R Ferguson. 1992. Dynamic method of test data generation for distributed software. *Information and Software Technology* 34, 8 (1992), 523 – 531. https://doi.org/10.1016/0950-5849(92)90146-G

[24] Kaituo Li, Christoph Reichenbach, Christoph Csallner, and Yannis Smaragdakis. 2012. Residual Investigation: Predictive and Precise Bug Detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 298–308. https://doi.org/10.1145/2338965.2336789

[25] Aditya P. Mathur. 2008. *Foundations of Software Testing* (1st ed.). Addison-Wesley Professional.

[26] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology* 14, 1 (2005), 1–41. https://doi.org/10.1145/1044834.1044835

[27] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

[28] Oracle. [n. d.]. HPROF Binary Format. https://java.net/downloads/heap-snapshot/hprof-binary-format.html

[29] Alex Potanin, James Noble, and Robert Biddle. 2004. Checking Ownership and Confinement: Research Articles. *Concurrency and Computation: Practice & Experience - Formal Techniques for Java-like Programs* 16, 7 (June 2004), 671–687. https://doi.org/10.1002/cpe.v16:7

[30] Easwaran Raman and David I. August. 2005. Recursive Data Structure Profiling. In *Proceedings of the 2005 Workshop on Memory System Performance (MSP '05)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/1111583.1111585

[31] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2002. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. In *Proceedings of the 16th International Conference on Supercomputing (ICS '02)*. ACM, New York, NY, USA, 274–284. https://doi.org/10.1145/514191.514229

[32] Barbara G. Ryder. 2003. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *Proc. of the 12th International Conf. on Compiler Construction (CC '03)*. Springer, 126–137. https://doi.org/10.1007/3-540-36579-6_10

[33] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[34] Micha Sharir and Amir Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program flow analysis: theory and applications*, Steven S. Muchnick and Neil D. Jones (Eds.). Prentice-Hall, Inc., Englewood Cliffs, NJ, Chapter 7, 189–233.

[35] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69. https://doi.org/10.1561/2500000014

[36] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 17–30.

[37] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 485–495. https://doi.org/10.1145/2594291.2594320

[38] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Alias Analysis for Object-Oriented Programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer Berlin Heidelberg, 196–232. https://doi.org/10.1007/978-3-642-36946-9_8

[39] Mana Taghdiri and Daniel Jackson. 2007. Inferring specifications to detect errors in code. *Automated Software Engineering* 14, 1 (2007), 87–121. https://doi.org/10.1007/s10515-006-0005-x

[40] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET. In *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings (Lecture Notes in Computer Science)*, Bernhard Beckert and Reiner Hähnle (Eds.), Vol. 4966. Springer, 134–153. https://doi.org/10.1007/978-3-540-79124-9_10