Heaps'n Leaks: How Heap Snapshots Improve Android Taint Analysis

Manuel Benz
Department of Computer Science
Paderborn University
Germany
manuel.benz@upb.de

Erik Krogh Kristensen
Department of Computer Science
Aarhus University
Denmark
erik@cs.au.dk

Linghui Luo
Department of Computer Science
Paderborn University
Germany
linghui.luo@upb.de

Nataniel P. Borges Jr.
CISPA Helmholtz Center for Information Security
Germany
nataniel.borges@cispa.saarland

Eric Bodden
Paderborn University & Fraunhofer IEM
Germany
eric.bodden@upb.de

Andreas Zeller
CISPA Helmholtz Center for Information Security
Germany
zeller@cispa.saarland

ABSTRACT

The assessment of information flows is an essential part of analyzing Android apps, and is frequently supported by static taint analysis. Its precision, however, can suffer from the analysis not being able to precisely determine what elements a pointer can (and cannot) point to. Recent advances in static analysis suggest that incorporating dynamic heap snapshots, taken at one point at runtime, can significantly improve general static analysis. In this paper, we investigate to what extent this also holds for *taint* analysis, and how various design decisions, such as when and how many snapshots are collected during execution, and how exactly they are used, impact soundness and precision. We have extended FlowDroid to incorporate heap snapshots, yielding our prototype Heapster, and evaluated it on DroidMacroBench, a novel benchmark comprising real-world Android apps that we also make available as an artifact. The results show (1) the use of heap snapshots lowers analysis time and memory consumption while increasing precision; (2) a very good trade-off between precision and recall is achieved by a *mixed mode* in which the analysis falls back to static points-to relations for objects for which no dynamic data was recorded; and (3) while a *single* heap snapshot (ideally taken at the end of the execution) suffices to improve performance and precision, a better trade-off can be obtained by using multiple snapshots.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

points-to analysis, heap snapshot, taint analysis, Soot

ACM Reference Format:
Manuel Benz, Erik Krogh Kristensen, Linghui Luo, Nataniel P. Borges Jr., Eric Bodden, and Andreas Zeller. 2020. Heaps'n Leaks: How Heap Snapshots Improve Android Taint Analysis. In *ICSE '20: International Conference on Software Engineering, May 23–29, 2020, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

ICSE '20, May 23–29, 2020, Seoul, South Korea
2020. ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Android is the world's most popular mobile operating system. Its official marketplace, Google Play Store, holds more than 3.3 million apps, which can be installed on billions of devices. To perform their tasks, apps frequently interact with sensitive information—from private images to banking details. Research shows that security-related bugs introduced by developers frequently put this sensitive information at risk [5, 6, 9, 27].

To identify such sensitive information leaks, *taint analysis* detects potential leaks by determining if data acquired on a sensitive source reaches a sink, where the information would no longer be secure. Such taint flows can be detected statically or dynamically. A *static* taint analysis, which we focus on in this paper, reasons about all possible execution paths in a program and aims to achieve (close to) perfect recall, i.e., it seeks to identify virtually all potentially sensitive information leaks. Static analyses, though, often suffer from a trade-off between accuracy and scalability. Although existing taint analysis tools such as FlowDroid [1] can be configured to conduct a relatively precise flow, context, and field-sensitive analysis, such configuration needs to be identified by possibly inexperienced users—and imprecise configuration causes the taint analysis to report substantial amounts of false positives [22].

A recent approach by Grech et al. addresses this problem by extending static pointer analysis with information extracted from *heap snapshots*, collected at runtime. As the authors show, one can improve soundness [10] by augmenting statically computed points-to information with *additional* data from the heap snapshots. Conversely, one can improve precision by *restricting* static points-to computation to such information present in the heap snapshots [11].

In their experiments, Grech's idea has been proven to be very effective; however, the utility of the technique is still vastly unexplored, leaving many questions unanswered. Both their implementation and evaluation are limited to pure pointer analysis only, but *how does using heap snapshots affect complex client analysis?* Furthermore, their setup was limited to collecting only a single heap snapshot for each execution of the analyzed application, yet *does a single heap snapshot suffice or are multiple heap snapshots better? When should they be collected?* Last but not least, Grech et al. considered the two extremes of augmenting and restricting points-to information based on heap snapshots, as well as proposed a recall-oriented blended analysis mode where they enhance a static model with dynamic information. It is an open question, *how to model the middle ground between imprecise static heap models and unsound dynamic heap snapshots?*

In this work, we present an empirical study in which we seek to reproduce the original experiments revised by Grech et al. but also go significantly beyond them to address these open questions. We make the following original contributions:

Using heap snapshots for Android taint analysis. We investigate how heap snapshots impacts the soundness and precision, not just of simple pointer analysis, but of a concrete client analysis, a *static Android taint analysis.*

Assessment of design decisions. We investigate how various essential design decisions impact precision and soundness of the analysis. In particular, we evaluate the impact of two novel extensions:

- information not only from a *single* heap snapshot but *multiple ones,* e.g., collected at various times during the execution; and
- *dynamic* heap models collected at runtime (precise, but possibly unsound) versus pure *static* heap models (sound, but possibly imprecise) versus *mixed* models that seek to define a sensible middle ground between those two extremes by focusing on precision and enhancing a dynamic model with static information.

Implementation and Benchmark. To evaluate the above decisions, we implemented *Heapster*, an extension to FlowDroid that can incorporate heap dumps. Additionally, we created *DroidMacroBench*, a set of 12 real-world Android applications that we manually labeled with ground truth for taint analyses.[1]

Evaluation. We explore the impact of different design decisions about *when to collect and how to consume heap snapshots.* In our evaluation we show that:

- adding heap snapshots can significantly improve the precision of taint analysis (from 50.3% to up to 94.7%);
- while restricting points-to information to that of the heap snapshots offers high precision it significantly harms recall. Our mixed mode solution, however, provides both good precision (77.1%) and good recall (68.4%). Its F1 score is the highest among all configurations;
- in all evaluated scenarios, incorporating heap snapshots significantly lowers the amount of computational resources required by the taint analysis, moreover, in 90% of the scenarios it also improves the analysis performance; and
- while a single heap snapshot, taken at the end of the runtime, suffices to significantly increase the analysis precision, additional snapshots, taken at different times, are beneficial for the analysis recall, achieving the best overall F1 score.

The remainder of this paper is organized as follows: In Section 2 we discuss limitations of the static analysis and how dynamic information can be used to enhance it. We then present Heapster, which uses heap snapshots to improve taint analysis, in Section 3 and in Section 4 our experiments showing the impact of different design decisions on the taint analysis results. After discussing threats to validity and related work in Sections 5 and 6, we close with the conclusion and future work.

2 BACKGROUND

2.1 Unsoundness in static analysis

An optimal static analysis would be both sound and precise, that is, it would report *all* relevant program information and would report only information that is, in fact, correct. A points-to analysis, for instance, is considered sound, if it reports that an allocation site (e.g. a new-expression) a is contained in a points-to set $pts(v)$ of a variable v in *every* case in which the program under analysis can assign to v an object allocated at a. A points-to analysis is maximally precise if it reports such a points-to relation *only* under those circumstances.

Points-to analysis is, however, known to be an undecidable problem, as of other static analyses. Much previous research has attempted to find trade-offs that are optimal for various applications, some experimenting with different context-sensitivities [16, 30, 32] and others selectively applying context-sensitivity [19, 35]. While these approaches are, to some extent, helpful in general, they are restricted by their limited ahead-of-time knowledge about the program under analysis and its actual execution.

In general, even the work that attempts to counter unsoundness so far has always been incomplete [21] due to complicated usage of reflection inside some programs. In such scenarios a sound model of these reflective features could cause a massive drop in performance, precision or both. While Smaragdakis et al. [31] showed that one could achieve provably sound results for parts of a program under analysis, the general undecidability makes the general problem unsolvable. Most work in the area of point-to analysis of Java thus attempts to reach some middle ground, where most features of the language are modeled soundly, while a small part of the language is modeled unsoundly.

2.2 Blended analysis

A blended analysis [4] is a static analysis that incorporates dynamic information obtained while executing the analyzed program. While dynamic information of various kind can be obtained and incorporated into the static analysis, in the following, we focus on heap snapshots.

Heap snapshots. A heap snapshot is a representation of objects from a running program, with the values and relationships that the objects had at the specific point in time in which the snapshot was collected. Many different industrial language/runtime implementations directly support collecting heap snapshots with little or no performance overhead [10, 20, 24, 33].

Heap snapshots are commonly used for debugging applications or finding memory leaks and are sometimes even collected from running production servers [26]. Besides being useful for debugging purposes, heap snapshots can be used to augment static analysis, since they provide a 100% accurate view of a single concrete heap. Uses of this concrete heap data include skipping parts of the analysis [7, 13] and improving the abstract heap precision in an analysis [11].

[1] Because DroidMacroBench comprises closed-source apps we cannot legally make it available as open source. We will make it available to other researchers, though, through a password-protected website. DroidMacroBench will also be made available to the artifact evaluation.

Even if enhanced with a complete picture of the heap, a static analysis must still perform approximations. Heap snapshots provide information only about objects in the heap. They lack information, e.g., about stack variables. Also, a heap does not allow the static analysis to determine which branches will be taken during runtime. Finally, given that heap snapshots only contain information about objects that existed at a particular point in time, the snapshot might fail to capture short-lived objects that only exist during a brief computation. These temporary objects are especially hard to capture given that some runtimes perform a garbage collection before creating a heap snapshot.

3 APPROACH

We designed a blended taint analysis *Heapster*, where we use heap snapshots as an upper bound in the static points-to analysis. We, therefore, obtain points-to sets that are strict subsets of the ones obtained through a purely static analysis. As our results show, this leads to a more precise analysis than using a strictly static heap model. However, by design, this also means that Heapster may sacrifice some level of soundness (or recall), i.e., Heapster may miss reporting some true positives. Yet, as our evaluation shows, useful tradeoffs do exist that keep recall high while nonetheless increasing precision significantly, resulting in an F1 score higher than that of purely static analysis.

To obtain such tradeoffs, it is crucial to understand and appreciate that the incorporation of heap snapshots is not just a binary "yes or no" decision. It opens up a design space within which one can make a number of sensible design choices—from how to use the heap snapshots to when to collect them—all of which affect the trade-off between the analysis' precision and soundness.

We implemented Heapster on top of FlowDroid [1], the leading tool for static taint analysis of Android applications. FlowDroid itself is built on top of the program-analysis framework Soot [14] and uses Soot's points-to analysis engine SPARK [15]. Thus, in our quest for improved precision and performance in FlowDroid, we modify SPARK to incorporate information from heap snapshots.

To explain our approach, we first present the inference rules used by the heap analysis in SPARK, and we then explain our incorporation of heap snapshots as modifications to these inference rules. We only present the rules for object allocation, field stores, field reads, and variable assignment, as these rules are the only relevant rules for our approach. Adding, e.g. context-sensitivity or method calls is orthogonal to our approach. Also, while our implementation was applied to Android, our modifications contain no Android-specific code; thus the same implementation can also be used to analyze Java applications.

Figure 1 depicts these inference rules. The following syntax is used in the rules: O_i denotes a concrete object, where *new* O_i creates a new object. v denotes a variable. $x.f$ denotes a field access of the field f of variable or object x. $l := r$ denotes an assignment from r to l. The "$\rightarrow$" symbol denotes a relation where the left side is a pointer, i.e., a variable or field reference, and the right side is the object the pointer points to. E.g., $v \rightarrow O_1$ and $v \rightarrow O_2$ denotes that the points-to set of v includes $\{O_1, O_2\}$. The analysis works by initializing all points-to sets as empty, and then applying the rules, thereby adding allocation sites to the points-to sets, until it reaches a fixed point.

$$(\textit{Store})\ \frac{o.f := v \quad o \rightarrow O_1 \quad v \rightarrow O_2}{O_1.f \rightarrow O_2} \qquad (\textit{Alloc})\ \frac{v := new\ O_i()}{v \rightarrow O_i}$$

$$(\textit{Read})\ \frac{v := o.f \quad o \rightarrow O_1 \quad O_1.f \rightarrow O_2}{v \rightarrow O_2} \qquad (\textit{Assignment})\ \frac{v := t \quad t \rightarrow O_1}{v \rightarrow O_1}$$

Figure 1: The four inference rules related to the heap used by the pointer analysis SPARK in FlowDroid

In the following we explain three different designs that differ in the way in which they incorporate the dynamic heap information into SPARK's static pointer analysis.

3.1 Using Heap Snapshots in Static Analysis

To incorporate heap snapshots into the points-to analysis, we modify the *Store* rule in Figure 1 to only apply if the same relation can be found in a given heap snapshot.

Grech et al. [11] instead remove the *Store* rule and modify the *Read* rule such that it directly reads points-to sets from their heap snapshot, Grech et al. [11] can thereby skip the static heap analysis altogether and achieve incredible performance. However, their approach leaves little room for modification. By modifying the *Store* rule, our approach makes the heap analysis in FlowDroid more precise, without entirely skipping it. We can thereby later selectively reenable the full heap analysis in FlowDroid, which we will do later in this section.

Loading our heap snapshots into the analysis exposes two new functions: *Heap* and *Field*. *Heap* is a function that accepts an allocation site and returns a set of heap objects, and *Field* is a function that, given a heap object and a field name, returns an allocation site or `null`. These two functions trivially work with multiple heap snapshots. Particularly, *Heap* will return the union of all the heap objects, that have been allocated at the given allocation site, from the heap snapshots.

Heapster treats the heap snapshots as being a ground truth, and this is achieved by restricting the statically computed points-to sets to contain only allocation sites also found in the heap snapshots. This can be expressed by replacing the *Store* rule from Figure 1 with the one presented in Figure 2[2]

$$(\textit{Store-d})\ \frac{o.f := v \quad o \rightarrow O_1 \quad v \rightarrow O_2 \quad h_i \in Heap(O_1) \quad Field(h_i, f) = O_2}{O_1.f \rightarrow O_2}$$

Figure 2: Replacement *Store* rule for SPARK

Generally, in Heapster, it is possible to derive points-to information from multiple heap snapshots gathered throughout the target application's runtime. We propose three variants to consume heap snapshots during static analysis: (1) *separate-heaps*, in which we restrict the static points-to information to what is present in a single heap snapshot, and run the analysis once for each snapshot given. (2) *merged-heaps*, which acts like separate-heaps but with all given heap snapshots used together in one single analysis run. (3)

[2] Note that = is used as equality in this rule.

static-fallback, where the heap snapshots are used in combination with the abstract heaps.

separate-heaps. This variant essentially computes taint flows for a single realistic application state. To nonetheless obtain a broader picture of the apps' behavior, we execute the analysis once for each heap snapshot collected and merge the findings as produced by FlowDroid. The separate-heaps configuration assures that no points-to information is merged among different heap snapshots, and thus eliminates yet another source of imprecision. This is because merging different states of an app's heap can lead to pointer combinations that are not feasible in any real execution.

Since the whole taint analysis and all its pre- and post- analyses are executed for each snapshot, the overall runtime of this variant is expected to be comparatively long, even though a single run should, in theory, be faster than with the other variants due to having the least pointer information available at a single time.

Taint-flows are merged by simply aggregating flows with equal source-sink pairs after all analysis runs have completed.

merged-heaps. This variant is conceptually similar to the *separate-heaps* one. Also here we restrict the points-to information to what is present in the given heap snapshots. In contrast to separate-heaps, however, the analysis is executed a single time with a set of heap snapshot merged together as the single source of points-to information.

Since only one analysis run is required, the analysis usually terminates faster. As previously stated, however, merging the findings of different heap snapshots might lead to an over-approximation due to generating unrealistic permutations of points-to information between multiple pointers.

static-fallback. There are two situations for which the *separate-heaps* and *merged-heaps* variants can cause the points-to set for an object field $O_1.f$ to be empty: (1) The heap snapshot only contains such objects allocated at O_1 for which the f field is *null*, or (2) the heap snapshot contains no objects allocated at O_1 at all. The latter case can be caused by objects of type O_1 never being allocated, or by the dynamic exploration failing to explore that part of the app, or because the heap snapshot—by chance—did not contain a short-lived O_1 object. Depending on the relevance of $O_1.f$ in the analyzed program, the analysis might erroneously produce wildly unsound points-to sets, which, due to the call graph becoming smaller, can result in large parts of the application never being analyzed.

This "mixed" analysis mode addresses these limitations with a middle-ground approach. We reuse the heap snapshots from the merged-heaps approach, but we no longer consider the heap snapshot as the single source of points-to information. In this approach, we use the static points-to information as a fallback option for fields that do not exist in the merged snapshot.

Our approach, just as Gretch et al. combines dynamic information extracted from heap snapshots with static analysis. However, while they enhanced a static points-to model with dynamic information to increase the soundness of the analysis, we follow the opposite direction, that is, we enhance the dynamic heap model with static points-to information to increase precision by only falling back to static points-to information where the dynamic is missing.

This approach introduces the following changes to SPARK's inference rules: We use the *Store-d* rule from the Section 3.1 when the *Heap* function returns a non-empty set of heap objects, otherwise we use the *Store* rule from Figure 1. We achieve this behavior by replacing the *Store* rule from Figure 1 with the *Store-d* rule from the Figure 2 plus the one from Figure 3.

$$(\textit{Store-sf})\ \frac{o.f := v \quad o \rightarrow O_1 \quad v \rightarrow O_2 \quad Heap(O_1) = \emptyset}{O_1.f \rightarrow O_2}$$

Figure 3: Additional *Store* rule for SPARK

In comparison to the merged-heaps and separate-heaps approaches, static-fallback forces only such points-to sets to be empty whose base object is indeed contained in the heap snapshot, and where the specific field was observed to have the value `null`. Specifically, static-fallback retains static points-to sets for such objects that the heap snapshot gives no information about.

Note that we decided to not run the experiments with separate heaps in combination with static fallback. Each snapshot will have some area of the app that is uncovered, and where the analysis therefore falls back to static analysis. Thus, the combination would probably behave close to the fully static approach due to combining all the static pointer information for the uncovered parts of each snapshot. Additionally, considering the huge amount of experiments to conduct for all apps, snapshots and heuristic, this would lead to an enormous time effort which is out of scope for this work.

3.2 Collecting Heap Snapshots

Heap snapshots can be collected at distinct points of app execution and with different frequency, opening up a virtually infinite set of design possibilities. They can be collected at regular intervals or at specific points of execution.

In this work, we chose to acquire heap snapshots after an input event was triggered. While an event could be triggered by the user, by clicking on the screen for example, or by the system, by receiving a message, we restricted our snapshot collection to user events, so that it could be executed on any stock Android device. We take snapshots after the app has finished its initialization and after each user action, e.g., clicks, text input, etc. For this, we ensure the UI is stable and the app is waiting for user input. From a set of heap snapshots we propose the following heuristics to select which ones to use during static analysis:

first. Uses only the snapshot acquired immediately after the app starts with no input given. With this heuristic, it is possible to automatically acquire heap snapshots for any app, without additional use of test generators or a manual app exploration.

last. Uses only the last snapshot taken during the app execution. The rationale behind this heuristic is that by interacting with the app new elements are allocated into the heap and, by collecting the last element, the heap would contain more information than collecting only the first one.

all. Uses all collected snapshots during the analysis. Since we take snapshots after each user action executed on the phone, this heuristic is equivalent to picking a snapshot for each user action.

unique-activity. Use a single snapshot for each activity. By extracting meta-data when taking a snapshot, such as the current user interface state, we can map snapshots to activities after collecting the snapshots. If multiple snapshots were collected for the same activity, *only the last one is chosen.* We opted for the last snapshot as it is more likely to contain the most substantial amount of points-to information, due to previous user interactions with the activity.

4 EVALUATION

We used Heapster to evaluate the effect of using heap snapshots for taint analysis of Android applications. Specifically, we aim to answer the following research questions:

RQ1: How are the precision and soundness of FlowDroid impacted by using heap snapshots as an upper bound for the static analysis?
RQ2: How do precision and soundness change when using the static-fallback approach?
RQ3: How often and when should heap snapshots be collected to gain a useful level of precision and soundness?
RQ4: How does using heap snapshots impact the runtime performance and memory consumption of the taint analysis?

4.1 Experimental Setup

Hardware Setup. We performed all experiments on a virtual machine with 4 CPU cores from an Intel Xeon E5-2695 v3 CPU and 100GB of RAM of which we assigned 80GB as Java virtual machine heap space.

Dataset. For our experiments, we chose the 200 most downloaded apps from the Google Play Store according to https://www.appbrain.com/apps/popular/.[3] We opted for large real-world applications for two major reasons: first, they resemble the most realistic use-case scenario for Heapster, as we expect it to be best suited for large applications with hard to analyze code. Second, we were unable to find an existing benchmark collection containing taint flows in real-world apps where we were able—and willing[4]—to execute the apps on our Android phones.

We then filtered the apps for which we could obtain heap snapshots. On a stock Android it is not possible to extract heap snapshots from a non-debuggable app, that is, where its `AndroidManifest.xml` file does not contain `debbugable=true`. To enable the acquisition of heap snapshots we unpacked, updated and re-packed all 200 downloaded apps using Apktool.[5] We then proceeded with the 143 apps ($\approx 72\%$) which could be successfully re-packed.

We then filtered out apps which did not contain any taint flows. For that, we analyzed the 143 re-packed apps with an unmodified version of FlowDroid, configured with its default list of sources and sinks, which resulted in taint flows being identified in 78 apps.

Finally, we filtered out apps that we were unable to install and execute on a device successfully. For instance, some apps could not be used after re-packing due to security countermeasures, such as certificate and signature checks. Other, also popular apps did not have a launchable activity and were meant to be either integrated or invoked from other apps, such as *Samsung Push Service*[6]. Interestingly, one app in our dataset (YouCam Makeup) launched into a developer mode after re-packing. The app behaved as usual but included extra options such as `dump db`, `dump logcat`, and an option to switch from production to development backend. In the end, we were able to install, execute and obtain heap snapshots from 49 apps, which we used for the remaining of this evaluation. As we will explain later, we chose a subset of those 49 apps to establish our artifact DroidMacroBench.

App Exploration. To get a reasonably good coverage of the used apps, we manually explored all of them on a OnePlus 6 phone and collected heap snapshots after each conducted user action. We thereby tried to capture all reachable app functionality and created accounts to pass login screens if the apps asked for it.

Experimental Configurations. In our experiments, we aimed to evaluate not only *how different usages of the heap snapshots impact taint analysis* but also *when should the heap snapshots be collected.* We analyzed every app in 11 different configurations comprised of all sensible permutations of the analysis mode and the snapshot acquisition time (c.f. Section 3.2). For each analysis, we used FlowDroid—or our extended version Heapster—with its default parameters. We opted for the default configuration since it is the most widely accepted baseline and allows direct comparisons to previous work that is also based on FlowDroid. Table 1 presents the different experimental configurations—including a traditional static analysis, without any heap snapshot—alongside the average and the accumulated number of snapshots over all analyzed apps.

Mode	Heuristic	∅ #Snapshots	∑ #Snapshots
full static	**none**	0	0
merged-heaps	**all**	48	2353
	unique-activity	10	483
	first	1	49
	last	1	49
separate-heaps[7]	**all**	48	2353
	unique-activity	10	483
static-fallback	**all**	48	2353
	unique-activity	10	483
	first	1	49
	last	1	49

Table 1: Average and accumulated number of heap snapshots used per configuration

Runtime restriction. Since a single analysis run of some of the larger apps took multiple hours to complete, we limit the runtime of the separate-heaps approach to a maximum of two times the runtime of the pure static approach. This prevents the separate-heaps approach from taking multiple days to complete when several snapshots are used for those apps. To still leave a chance for the separate-heaps approach to complete on small apps that do not take as much time, we also allow it to run at least one additional hour to what the pure static approach needed.

[3] The full list of apps is available at https://goo.gl/7NwTMu

[4] Annotated taint flows for Android apps are provided for known malware.

[5] https://ibotpeaches.github.io/Apktool/

[6] https://play.google.com/store/apps/details?id=com.sec.spp.push.

[7] We omit *first/last* for the *separate-heaps* analysis mode because it is equivalent to use *first/last* with merged-heaps mode.

4.2 DroidMacroBench

As one contribution of this work, we provide DroidMacroBench, a benchmark of 12 of the 200 top most downloaded *real-world* Android applications with labeled taint flows. Similar to the popular DroidBench [1] micro benchmark, also DroidMacroBench provides Android applications labeled with ground truth for taint analyses. However, in contrast to DroidBench, DroidMacroBench comprises real-world Android applications which are, in essence, much larger, more complex and altogether more realistic benchmarks for analysis at hand.

Using DroidBench for assessing Heapster was not an option since its apps are not executable. Moreover, even if they were, the DroidBench apps are too small and trivial to cause a realistic static over-approximation which Heapster aims to counter. Furthermore, due to the missing complexity, such small apps are not able to profit from the performance gains possible when using heap snapshots.

To establish DroidMacroBench, we manually investigated and labeled FlowDroid's findings for 12 of the 49 apps as feasible/infeasible taint flow. Investigating a large number of apps manually would be a worthwhile future endeavor but due to the significant amount of manual labor involved would probably have to be conducted as a community effort. By exposing DroidMacroBench we hope to seed such an effort.

We classify a taint flow as infeasible if during a manual investigation we find it impossible for data from a given source to flow to a given sink. This manual investigation was done by having one author initially classify a taint flow as produced by FlowDroid, and this classification was then independently confirmed by another author. In the remainder of this paper, we consider reports of feasible flows as true positives, other taint reports as false positives.

In our classification, of FlowDroid's findings, we label the reported taint flows as *feasible or infeasible*, but we make no determination as to what kind of data flows, and whether this data is sensitive. The high complexity of the code and in particular the frequently employed obfuscation made this determination impossible. To evaluate DroidMacroBench, however, this information is also not required.

DroidMacroBench comprises 12 real-world Android applications with a total of 199 taint flows found by FlowDroid, of which we were able to classify 157 into 79 feasible and 78 infeasible taint flows. For the unclassified flows, the taint flows are so complex that we were not able to classify them with the necessary certainty.

In the future, we plan to provide DroidMacroBench and the corresponding ground truth as a benchmark suite compliant to the ReproDroid [25] framework, to allow others to reproduce our experiments easily, but also to run their experiments on DroidMacroBench.

4.3 RQ1: How are the precision and soundness of FlowDroid impacted by using heap snapshots as an upper bound for the static analysis?

To answer this question, we used the apps from DroidMacroBench and analyzed them with both *separate-heaps* and *merged-heaps* approaches, which consider the heap snapshots as upper bounds for the static analysis. We used all the collected heap snapshots. We then compared these results with those from the benchmark's ground truth. Since we use real, commercial, applications, the total number of existing taint flows is unknown. For our analysis, we assume that the taint flows found by FlowDroid represent all possible taint-flows that can exist in the apps and we, thus, measure precision and recall *relative* to the findings of the purely static approach. This assumption is sensible since we seek to determine exactly to what extend the incorporation of heap snapshots allows one to change FlowDroid's original precision and recall curve. Our results are shown in Table 2.

Separate-heaps approach. The *separate-heaps* approach identified a total of 19 taint flows, 18 of which are feasible, resulting in a precision of 94.7%, compared to 50.3% precision with the purely static analysis.

The separate-heaps approach missed 61 out of 79 feasible taint flows found by the purely static analysis, resulting in a recall of 24.1%. This high precision/low recall trade-off resulted in an *F1 Score* of 0.37.

Merged-heaps approach. The *merged-heaps* approach identified 21 taint flows, of which 19 are feasible. That is, by merging multiple heap snapshots into a single one, the analysis was able to identify 2 new taint flows, 1 of which is a true positive, which were not identified by multiple single heap experiments. Overall, compared to *separate-heaps*, *merged-heaps* obtained marginally lower precision (-4.2%) and higher recall (+1.3%), resulting in an *F1 score* improvement of 0.01. While these values are pretty similar, the runtime performance of these approaches differ significantly, as we show further in our experiments.

These results suggest that, while restricting the points-to information to what was observed dynamically, leads to excellent precision, but at the same time the overall trade-off between precision and recall is rather drastic: both approaches show an F1 score smaller than that of the purely static approach (0.67). Nevertheless, given that developers tend to abandon tools with high false positive rates [3], this high precision/low recall trade-off may be adequate under specific scenarios.

Generally, different analysis scenarios require different trade-offs between precision and recall. In our work, we empirically explore which trade-offs the use of heap dumps offer. We leave to related work [29] to judge in which scenarios which trade-off is ideal.

> Using heap snapshots as an upper bound increases precision from 50.3% to up to 94.7%, but sacrifices at least 77.2% of recall. The use of multiple heap snapshots has limited effect.

Those results also motivate the static-fallback approach, which we evaluate next.

	Full Static		
	Classified as		
Input	True	False	Total
True	TP = 79	FN = 0	79
False	FP = 78	TN = 0	78
Total	157	0	157

Precision = 50.3% *Recall* = 100.0%
Accuracy = 50.3% *F1 Score* = 0.67

	Separate-heaps		
	Classified as		
Input	True	False	Total
True	TP = 18	FN = 61	79
False	FP = 1	TN = 77	78
Total	19	138	157

Precision = 94.7% *Recall* = 22.8%
Accuracy = 60.5% *F1 Score* = 0.37

	Merged-heaps		
	Classified as		
Input	True	False	Total
True	TP = 19	FN = 60	79
False	FP = 2	TN = 76	78
Total	21	136	157

Precision = 90.5% *Recall* = 24.1%
Accuracy = 60.5% *F1 Score* = 0.38

Table 2: Confusion matrix for static analysis and our merged-heaps and separate-heaps approaches using all collected heap snapshots as upper bound for analysis. Note: For Full Static, the recall is 100% by definition as we measure recall *relative* to it.

4.4 RQ2: How do precision and soundness change when using the static-fallback approach?

By construction, the static-fallback approach from Section 3.1 should produce strictly larger sets of taint flows compared to the merged-heaps approach and strictly smaller sets compared to the purely static one. It is thus interesting to see how these additional/missing flows impact precision and recall compared to the other approaches. To answer this question, we analyzed the apps from DroidMacroBench with the *static-fallback* approach and compared these results with the ground truth from DroidMacroBench and the results from our previous experiments. Table 3 shows the results of our experiments with the static-fallback approach when all heap snapshots were used and merged together.

Static-fallback detects a total of 70 taint flows in DroidMacroBench, of which 54 are feasible. Compared to separate-heaps and merged-heaps this represents 49 more taint flows detected, with 35 more feasible taint flows detected. Compared to the purely static approach, the static-fallback analysis identified 45% of all known taint flows and 68% of the feasible ones, with a precision of 77.1% (+26,8%) and a recall of 68.4% (-31.6%).

We conclude that due to the increased recall compared to the separate-heaps and merged-heaps approaches, and the increased precision compared to the purely static approach, static-fallback yields a very favorable trade-off between precision and recall with a F1 Score of 0.72, compared to the *merged-heaps*'s score of 0.38 and to the purely static one of 0.67 (even only 0.37 for separate-heaps).

	Static-fallback		
	Classified as		
Input	True	False	Total
True	TP = 54	FN = 25	79
False	FP = 16	TN = 65	81
Total	70	90	160

Precision = 77.1% *Recall* = 68.4%
Accuracy = 74.4% *F1 Score* = 0.72

Table 3: Confusion matrix for our static-fallback approach when all collected heap snapshots are used for analysis.

Heap snapshots with static-fallback offered the best trade-off between precision (77.1%) and recall (68.4%), with the highest F1 score overall (0.72).

4.5 RQ3: How often and when should heap snapshots be collected to gain a useful level of precision and soundness?

We discussed in Section 3 that using heap snapshots alongside static analysis is not a binary choice, but opens up different design decisions. In Sections 4.3 and 4.4 we evaluated how the way heap snapshots are used impact the analysis results. In this section our goal is to gather empirical evidence about the impact of *how often and when* heap snapshots are collected on the analysis findings. To evaluate this research question we measured how many feasible/infeasible taint flows were found with all possible Heapster configurations. The results of all experiments are shown in Figure 4 and summarized in Table 4. In Figure 4 each color represents a different app, and the same color applies to the same app throughout the chart.

From Figure 4 it is possible to see that using heap snapshots always improves precision and worsens recall, independently of how they are used. Nevertheless, the choice of which heap snapshots are used does, in fact, affect the results. Using only the snapshot of the *first action* appears to restrict the taint analysis, resulting in the lowest number of correctly detected taint flows in both merged-heaps (11) and static-fallback (47). The same behavior is also observed when using only the heap snapshot of the *last action*, however, with slightly better results—16 and 51 feasible flows detected.

Note that it might seem counter-intuitive at first, that for static-fallback/all, the analysis reports fewer findings than for static-fallback/unique-activity. After all, the former uses more heap snapshots than the latter. However, while for the configurations merged-heaps and separate-heaps it is, in fact, true, that there the "all" option yields the highest recall by definition, for static-fallback this does not hold. This is because, when adding a further snapshot in static-fallback, this new snapshot might well add a heap object for an allocation site where none of the previous snapshots contained objects for that allocation site. This then prevents the static fallback from falling back to the purely static analysis for this allocation site, hence showing a tendency to increase precision but lower recall.

Nonetheless, the recall generally benefited from using multiple snapshots. For the static-fallback approach, when compared to a single snapshot, using multiple heap snapshots improved the precision significantly, by more than 10%, with the best results overall—an F1 Score of 0.77—being achieved with unique-activity, where Heapster takes a snapshot after each activity.

Table 4 shows that in many cases using all snapshots yields similar results to only using the last snapshot for each activity. Since pointer information is only lost on garbage collection and when a field is re-assigned, similar performance of those two heuristics shows that this does not happen often, i.e., there is not much additional pointer information when taking all snapshots instead of only the last for each activity.

From Figure 4 it is also possible to observe that the results vary depending on which app is analyzed. When analyzed with the static-fallback approach with *first* and *last* heuristics, the apps `com.contextlogic.wish` and `com.bitstrips.imoji` exhibit the opposite trend as the one observed on the aggregated result, with both the apps finding more feasible taint flows with the *first* heuristic.

In conclusion, the experiments suggest that two different snapshot-collection strategies should be used, depending on what level of precision is desired. If precision is critical, a snapshot collected right as the app is started should be used as the only snapshot, as this achieved 100% precision with our merged-heaps approach. If precision is less critical, a snapshot should be collected for each activity in the app. This gives a good trade-off with both high precision and recall, and only requires an average of 10 collected heap snapshots on our benchmark applications.

> When and how often a heap snapshot is collected has an impact of up to 16.4% on the precision and 13% on the recall, with the best overall results being achieved with a single snapshot per activity and the static-fallback approach.

4.6 RQ4: How does using heap snapshots impact the runtime performance and memory consumption of the taint analysis?

A major reason for developers to use less precise static-analysis configurations is to improve scalability, i.e., to reduce the amount of time taken by the analysis and the amount of computational resources needed. In this section, we aim to measure how the use of heap snapshots impacts these factors.

Analysis duration. Figure 5 depicts the relative speedups in analysis time for of our 10 configurations across all 49 apps, with higher values representing faster analysis. The green line shows the median speedup, and the red line shows the average one. The time reported includes all parts of the analysis, including loading the heap snapshots into memory.

Regarding the analysis duration, the average analysis time for the purely static configuration was 1 hour and 3 minutes, while the average running time for merged-heaps/last (the fastest on average) was only 13 minutes.

separate-heaps. The separate-heaps configurations were slower for most of our applications, with the separate-heaps/all resulting in a slowdown factor of 27, the largest among all. The difference between the separate-heaps/all and separate-heaps/unique-activity

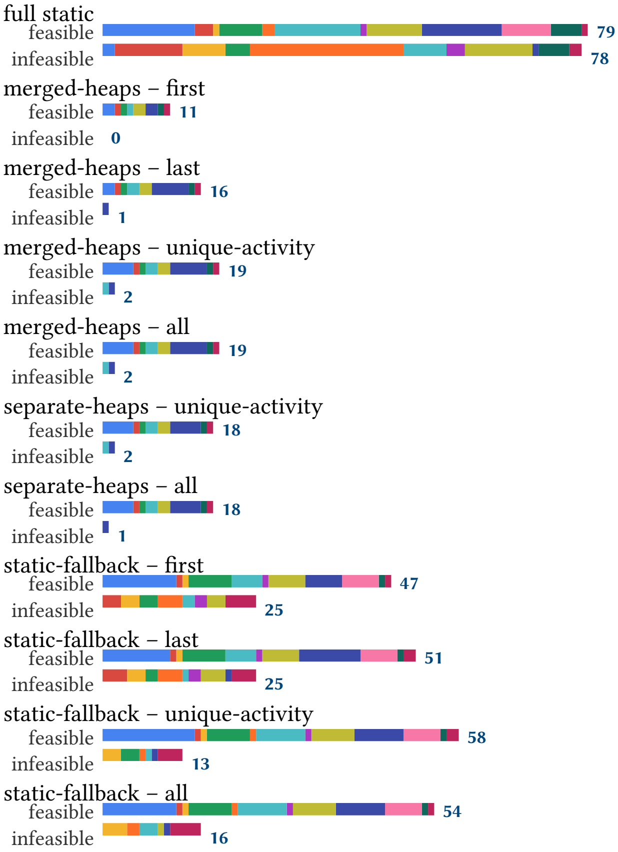

Figure 4: A barchart showing how many feasible/infeasible taint flows were found with each configuration. The individual apps are represented using different colors.

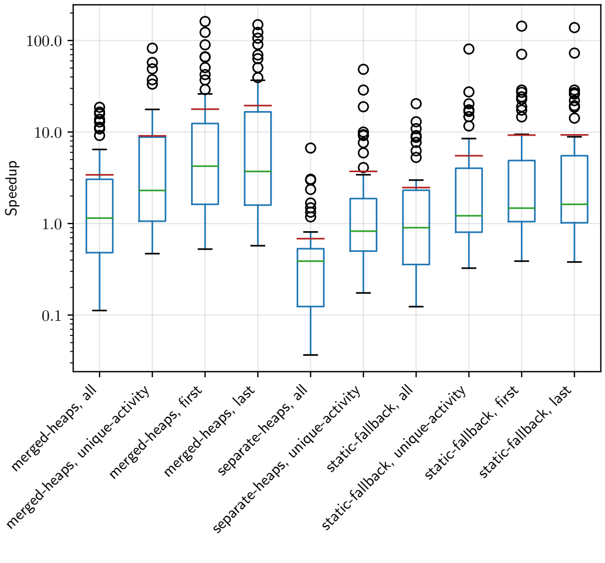

Figure 5: Execution time speedup compared to *full-static*

Mode	Heuristic	#TP	#FP	#TN	#FN	Precision	Recall	F1 Score
full static	**none**	79	78	0	0	50.3%	100.0%	0.67
separate-heaps	**unique-activity**	18	2	76	61	90.0	22.8%	0.36
	all	18	1	77	61	94.7%	22.8%	0.37
merged-heaps	**first**	11	0	78	68	100.0%	13.9%	0.24
	last	16	1	77	63	94.1%	20.3%	0.33
	unique-activity	19	2	76	60	90.5%	24.1%	0.38
	all	19	2	76	60	90.5%	24.1%	0.38
static-fallback	**first**	47	25	56	32	65.3%	59.5%	0.62
	last	51	25	55	29	67.1%	63.8%	0.65
	unique-activity	58	13	67	22	81.7%	72.5%	0.77
	all	54	16	65	25	77.1%	68.4%	0.72

Table 4: Aggregated true/false positives/negatives and relative recall/precision/F1-score per configuration.

configurations is entirely explained by the larger numbers of snapshots the analysis has to run on in the separate-heaps/all configuration.

With the separate-heaps approach the analysis timed out for 21 of the 98[8] experiments runs. The apps where the analysis timed out overlap with the apps for which we witnessed out-of-memory errors (which will be discussed later in this section). However, the analysis never timed out for any of the apps in DroidMacroBench.

This result shows that even though Heapster scales, it does not scale to the point where it is feasible to run the analysis repeatedly with many different heap snapshots.

merged-heaps. In Section 4.5 the merged-heaps and separate-heaps approach had very similar precision and recall results when using multiple heap snapshots. However, as can be seen in Figure 5, while the separate-heaps approach did not scale, the merged-heaps approach achieved by far the best performance, with the average speedup from the separate-heaps to the merged-heaps approach improving from 3.7 to 9.1 and 0.6 to 3.4 for the all and unique-activity configurations respectively.

Moreover, the performance of the approach is better if a smaller collection of heap snapshots are used. The best-scaling configuration on average is using only the heap snapshot collected as the application has started up, with an average speedup of 19.5 and max of 149.

This is not surprising, as the merged-heaps approach should—by construction—be the most scalable of all our evaluated approaches. The relatively worse performance of using multiple snapshots is also unsurprising, as these configurations produce larger points-to sets, and take longer loading the heap snapshots from the file system. Some apps experienced a performance slowdown when using the merged-heaps approach. These are apps where the analysis itself took very little time (down to about 5 seconds), and thus the overhead caused by loading the heap snapshots from the file system was significant.

static-fallback. The results for our static-fallback approach reveal performance gains between 3 and 10 times on average, with an analysis duration time between 31 and 39 minutes on average. These improvements are not as high as those obtained by separate-heaps and merged-heaps due to the many cases in which the analysis ends up inspecting parts of the app with no heap snapshots information.

Looking back at our F1 scores from Section 4.5, static-fallback with both multiple heap snapshots heuristics had comparable, and great, precision results. Given that the static-fallback/unique-activity approach also has better runtime performance, our experiments indicate that it presents the best trade-off.

Memory consumption. Figure 6 depicts the average memory usage by the analysis of our 11 configurations across all 49 apps. 103 out of the 539 runs (≈19%) could not be completed due lack of sufficient memory, even with the substantial 80GB heap space reserved for the JVM. This error happened on 13 different apps, none of which are contained in DroidMacroBench nor were used in the previous evaluations.

On average the entirely static analysis, without any heap snapshot required the most substantial amount of memory, 44.29GB. The separate-heaps and merged-heaps approaches required the least amount of memory (18.2GB) when using the only the last collected snapshot, with the separate-heaps approach requiring the lest overall amount of memory (≈23.13GB)[9].

The static-fallbackapproach required more memory than both separate-heaps and merged-heaps when using the same heuristic. However, it still required more than 10% less memory then its entirely static counterpart. Considering the static-fallbackapproach with the *unique-activity* heuristic, which achieved the highest F1 score in our previous evaluations, it led to an average reduction of ≈27% in memory consumption.

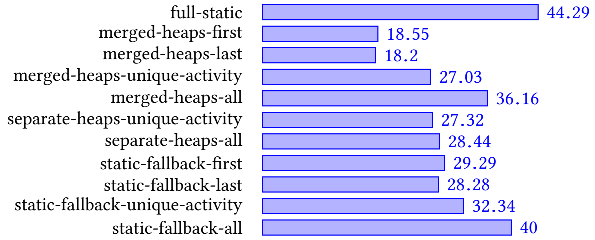

Figure 6: Average Memory (GB) for each experiment configuration

[8] separate-heaps/all and separate-heaps/unique-activity across 49 apps.

[9] The separate-heaps and merged-heaps approaches are equivalent when using the *first* or *last* snapshot heuristics, thus for the average memory consumption we them on both approaches.

If running time/scalability is a concern, then merged-heaps configurations should be chosen, as it requires significantly less time and resources. static-fallback achieved the best trade-off between precision, recall and runtime performance, reducing analysis time by ≈ 47% and memory consumption by ≈27%.

5 THREATS TO VALIDITY

We here describe four limitations that might threaten the validity of the results we have presented.

Non-deterministic taint flows. During our experiments we noticed that FlowDroid's results are non-deterministic, i.e., two runs of FlowDroid for the same app with the same configuration might produce different results. We clarified with FlowDroid's authors that this is a known bug in a post-analysis component that reconstructs context-sensitive path information for an identified taint flow.

It remains unfixed in the latest version and it is not planned to get fixed[10]. The bug makes our ground truth slightly uncertain, due to the ground truth being build from FlowDroid's findings. However, others using FlowDroid for their experiments also fell victim to the same non-determinism, yet we are the first reporting it.

In our experiments, this non-determinism can lead to unintuitive findings. In particular, all our analysis configurations refine the static points-to information, which means that —by construction— we should always find a subset of the taint flows found by the **full static** configuration. In our experiments, however, we sometimes encountered taint flows, that are not found by the **full static** configuration. For example in the `com.soundcloud.android` app 2, respectively 4, taint flows were not found with the **full static** configuration but were found when using the **static fallback** approach with the first and last snapshot respectively. (No other configuration had such taint flows for this app.) In the evaluation, we limit the results to only consider those we had found as being feasible/infeasible with the **full static** configuration.

102 of the total 574 taint flows (including unclassified ones) found across 120 experiment runs[11] were false positives not found by the **full static** configuration (≈ 0.85 per run).

We compared the findings to some of the apps in DroidMacroBench and found that the deviance comprises not more than a handful of findings per run and that the distribution of false to true positives among the deviating findings is random but corresponds to the overall distribution of false to true positives in all findings. Hence, the overall findings remain valid.

App selection. The chosen apps for building the ground truth might not be representative for all Android applications. The selection of apps for manual investigation was arbitrary among all the evaluated apps. However, the selection was not influenced by any a priori knowledge of taint flow feasibility, since feasibility was only assessed thereafter.

User action order and code-coverage might influence heap-data. Our current experiments only measure how user interaction influences the heap-data and, therefore, the analysis results in general (section 4.5). We do not meassure in which manner the order of user actions or the amount of covered code influences the heap-data and the analysis results. To study the correlation between user-action-order or coverage and analysis performance, we have to design our experiment differently by exploring the apps aiming for these metrics. Obviously there is also a tradeoff: adding two additional dimensions to the experiments leads to many more experiment permutations and the payoff may be unclear. Nonetheless, this is an interesting research question that could be addressed in the future.

Feasible labeled flows might not be realizable during runtime. We only checked data-flow of the found taint-paths for feasibility assessment. We were not able to make a 100% confident assessment of control-flow feasibility. Hence, it is possible that some of the flows labeled as feasible cannot be realized in any dynamic execution. In such cases, the recall given for the heap snapshot approaches might actually be higher than reported.

6 RELATED WORK

Taint analysis has been primarily used on Android applications to discover security and privacy issues. According to Li et al.'s literature review [18], 46 approaches using static taint analysis focus on detecting private data leaks in Android apps. These approaches achieved good precision by regarding multiple analysis sensitivities from context-, flow-, field-, object- and path-sensitivity when applied to artificial benchmarks [1, 8, 17, 34]. However, very few of them evaluated real-world applications due to the lack of ground truth. Studies have shown that static tools are most likely to be adopted by developers when they yield a low rate of false positives [3, 12]. Thus, it is essential to know how good these tools perform when applied to real-world applications. In our paper, we evaluated both a static (FlowDroid) and our hybrid approach (Heapster) with top Android applications from the Google Play Store. Our results show, in comparison to FlowDroid, that using heap snapshots always improves the precision in all configurations we tried. The measured F1-scores also suggest that our mixed-mode static-fallback configuration can achieve a better trade-off between precision and recall than other configurations that extend or restrict points-to computations in all cases (see Table 4).

In the area of dynamic analysis for Android applications, TaintDroid [5] is one of the most prominent tools. However, TaintDroid was implemented as an extension to the Android platform (i.e., taint tracking within the Dalvik VM Interpreter) rather than a standalone tool, which makes it hard to maintain for every new version of Android (i.e., the last supported version was Android 4.3 in 2013). Our tool, instead, is built on top of FlowDroid, which is well maintained since its first appearance. We were able to apply our tool to the top most downloaded Android applications targeted to the recent Android versions (e.g. com.spotify.music was targeted for Android 8.0 and com.contextlogic.wish was for Android 8.1).

Hybrid approaches that combine static and dynamic analysis seem to become more promising in recent researches [11, 23, 28, 36, 37]. Dufour et al. were probably the first to propose "blended analyses" (and to coin this term), in which static analyses incorporate

[10] The authors are working on a complete overhaul of their taint propagation algorithm which will fix the problem. There is, however, no release scheduled yet.

[11] 10 configurations using heap snapshots across 12 benchmark applications results in 120 runs.

dynamic information information to optimize precision and/or recall [4]. The authors present a concrete example of a blended escape analysis. Our work is inspired by the blended/hybrid approach of Grech et al. [10, 11], in which they used heap snapshots to reduce the cost of modeling the heap in static analysis and achieved excellent scalability, precision, and recall according to their evaluation using the DaCapo Java benchmark suite [2]. In their approach, the heap snapshots were taken on JVM exit for each application [10]. To maintain more information (e.g. the actual linking between objects and references) which can be preserved in the heap snapshots, they used the standard JVM instrumentation agents to inject code into the applications. However, such enrichment method cannot be easily applied for Android, because no standard instrumentation agent exists. In addition, determining the optimal timing to collect heap snapshots is not trivial, since Android applications are no standalone applications, but rather plugins for the Android framework. They consist of multiple components such as activity, service, broadcast receiver, and content provider, which have distinct lifecycles, and often dozens of callbacks that respond to various user actions such as clicks or inputs. Our approach does not require any instrumentation of the Android applications being analyzed, but consider heuristics with regard to actions and activities in our experiments. Our results show that a good trade-off between precision and recall can be achieved by using snapshot collected at different times during application execution.

7 CONCLUSION

We showed that heap snapshots, collected from unmodified apps on a stock Android runtime, can be used to improve the performance and precision of taint analysis for Android. Our approach achieves this by modifying FlowDroid to use heap snapshots as upper bounds for the computed points-to sets during its static analysis.

We introduced a new novel middle-ground approach for using heap snapshots, which selectively falls back to a purely static analysis when the heap snapshots contain no information about an allocation site. Compared to a fully static analysis, we demonstrated that this middle-ground achieves the best trade-off between precision, recall, runtime performance and scalability.

We additionally investigated how multiple heap snapshots affect analysis results and performance. We observed that, in all cases, multiple snapshots lead to a better recall. Nevertheless, single heap snapshots, taken before closing the app, produced great runtime performance and very high precision, indicating that this approach may be more adequate for specific scenarios.

Finally, we introduced DroidMacroBench, a collection of 12 realistic Android apps, which are all among the top 200 most downloaded Android apps on the Google Play Store, along with annotated taint flows from FlowDroid annotated as being feasible/infeasible, and use these 12 apps for evaluating our approach.

In the future, we plan to provide DroidMacroBench and the corresponding ground truth as a benchmark suite compliant to the ReproDroid [25] framework, to allow others to reproduce our experiments easily, but also to run their experiments on DroidMacroBench. Additionally we plan to explore an iterative refinement of the static taint analysis with heap snapshots such that our approach could be used online, i.e, identify leaks which could happen given the current app state.

We hope that our work inspires more research into how heap snapshots can be used to augment a static analysis, and that DroidMacroBench becomes a starting point for Android static analysis developers to use more realistic apps in the evaluations.

Replicability: To facilitate replication and extension, our work is available as open source at: https://bit.ly/38F5tUH

8 ACKNOWLEDGEMENTS

This research was supported by the German Research Foundation (DFG) within the Finding and Demonstrating Undesired Program Behavior (TESTIFY) project and the research training group Human Centered Systems Security (NERD.NRW) sponsored by the state of North Rhine-Westphalia in Germany.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 259–269. https://doi.org/10.1145/2594291.2594299

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[3] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis : An Empirical Study. *ASE16* (2016), 332–343. https://doi.org/10.1145/2970276.2970347

[4] Bruno Dufour, Barbara G Ryder, and Gary Sevitsky. 2007. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 118–128.

[5] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones William. *Commun. ACM* 57, 3 (2014), 99–106. https://doi.org/10.1145/2494522 arXiv:1005.3014

[6] William Enck, Damien Octeau, Patrick D. McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association. http://static.usenix.org/events/sec11/tech/full_papers/Enck.pdf

[7] Asger Feldthaus and Anders Møller. 2014. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 1–16. https://doi.org/10.1145/2660193.2660215

[8] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*.

[9] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks, WISEC 2012, Tucson, AZ, USA, April 16-18, 2012*, Marwan Krunz, Loukas Lazos, Roberto Di Pietro, and Wade Trappe (Eds.). ACM, 101–112. https://doi.org/10.1145/2185448.2185464

[10] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2017. Heaps don't lie: countering unsoundness with heap snapshots. *PACMPL* 1, OOPSLA (2017), 68:1–68:27. https://doi.org/10.1145/3133892

[11] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2018. Shooting from the heap: ultra-scalable static analysis with heap snapshots. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018* (2018), 198–208. https://doi.org/10.1145/3213846.3213860

[12] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs? 672–681.

[13] Erik Krogh Kristensen and Anders Møller. 2017. Inference and Evolution of TypeScript Declaration Files. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Marieke Huisman and Julia Rubin (Eds.), Vol. 10202. Springer, 99–115. https://doi.org/10.1007/978-3-662-54494-5_6

[14] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*. http://www.bodden.de/pubs/lblh11soot.pdf

[15] Ondrej Lhoták and Laurie J. Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science)*, Görel Hedin (Ed.), Vol. 2622. Springer, 153–169. https://doi.org/10.1007/3-540-36579-6_12

[16] Ondrej Lhoták and Laurie J. Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings (Lecture Notes in Computer Science)*, Alan Mycroft and Andreas Zeller (Eds.), Vol. 3923. Springer, 47–64. https://doi.org/10.1007/11688839_5

[17] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 280–291. https://doi.org/10.1109/ICSE.2015.48

[18] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information & Software Technology* 88 (2017), 67–95. https://doi.org/10.1016/j.infsof.2017.04.001

[19] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 129–140. https://doi.org/10.1145/3236024.3236041

[20] Sheng Liang and Deepa Viswanathan. 1999. Comprehensive Profiling Support in the Java Virtual Machine. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems, May 3-7, 1999, The Town & Country Resort Hotel, San Diego, California, USA*, Murthy V. Devarakonda (Ed.). USENIX, 229–242. http://www.usenix.org/publications/library/proceedings/coots99/liang.html

[21] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

[22] Linghui Luo, Eric Bodden, and Johannes Späth. 2018. *A Qualitative Analysis of Taint-Analysis Results.* Technical Report. Heinz Nixdorf Institute, Paderborn University.

[23] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[24] József Mihalicza, Zoltán Porkoláb, and Abel Gabor. 2011. Type-preserving heap profiler for C++. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*. IEEE Computer Society, 457–466. https://doi.org/10.1109/ICSM.2011.6080813

[25] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 331–341. https://doi.org/10.1145/3236024.3236029

[26] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *ECOOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings (Lecture Notes in Computer Science)*, Rachid Guerraoui (Ed.), Vol. 1628. Springer, 116–134. https://doi.org/10.1007/3-540-48743-3_6

[27] Siegfried Rasthofer, Steven Arzt, Robert Hahn, Max Kohlhagen, and Eric Bodden. 2015. (In)Security of Backend-as-a-Service. In *blackhat europe 2015*. http://bodden.de/pubs/rah+15backend.pdf

[28] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques. In *Network and Distributed System Security Symposium (NDSS)*.

[29] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

[30] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

[31] Yannis Smaragdakis and George Kastrinis. 2018. Defensive Points-To Analysis: Effective Soundness via Laziness. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[32] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 387–400. https://doi.org/10.1145/1133981.1134027

[33] Pasquale Stirparo, Igor Nai Fovino, and Ioannis Kounelis. 2013. Data-in-use leakages from Android memory - Test and analysis. In *9th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications, WiMob 2013, Lyon, France, October 7-9, 2013*. IEEE Computer Society, 701–708. https://doi.org/10.1109/WiMOB.2013.6673433

[34] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. 1329–1341. https://doi.org/10.1145/2660267.2660357

[35] Shiyi Wei and Barbara G. Ryder. 2015. Adaptive Context-sensitive Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs)*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 712–734. https://doi.org/10.4230/LIPIcs.ECOOP.2015.712

[36] Michelle Y. Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/intellidroid-targeted-input-generator-dynamic-analysis-android-malware.pdf

[37] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. SmartDroid. *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices - SPSM '12* (2012), 93. https://doi.org/10.1145/2381934.2381950