Refuting Heap Reachability

Bor-Yuh Evan Chang

University of Colorado Boulder
bec@cs.colorado.edu

Abstract. Precise heap reachability information is a prerequisite for many static verification clients. However, the typical scenario is that the available heap information, computed by say an up-front points-to analysis, is not precise enough for the client of interest. This imprecise heap information in turn leads to a deluge of false alarms for the tool user to triage. Our position is to approach the false alarm problem not just by improving the up-front analysis but by also employing after-the-fact, goal-directed *refutation analyses* that yield targeted precision improvements. We have investigated refutation analysis in the context of detecting statically a class of Android memory leaks. For this client, we have found the necessity for an overall analysis capable of path-sensitive reasoning interprocedurally and with strong updates—a level of precision difficult to achieve globally in an up-front manner. Instead, our approach uses a refutation analysis that mixes highly precise, goal-directed reasoning with facts derived from the up-front analysis to prove alarms false and thus enabling effective and sound filtering of the overall list of alarms.

The depth and breadth of what static verification tools can do is really quite astounding. However, the unfortunate truth is that for a software developer to fully realize the benefits of static verification, she must go through the onerous task of triaging the warnings that such a tool produces to determine whether the warnings are true bugs or false alarms. Only then can bug fixes be brought to bear. The situation is particularly dramatic when we consider the verification of modern programs that make extensive use of heap allocation. Precise heap information is a prerequisite for effective reasoning about nearly any non-trivial property of such programs. While there is a large body of work on heap reasoning, including in broad areas like pointer analysis and shape analysis, the all-too-common situation is that the available heap information is not quite precise enough for the client of interest.

Our position is to approach the false alarm problem not just by improving up-front precision but also with analyses for alarm triage that can wield after-the-fact, targeted precision improvements on demand. In this context, we investigate the combination of a scalable, off-the-shelf points-to analysis as an up-front analyzer with an ultra-precise, after-the-fact refutation analysis for heap reachability queries. The particular challenge that we examine is how to maximally leverage the combination of the up-front analysis and the after-the-fact refutation analysis.

The development of our approach has been driven in part by building a tool for detecting a pernicious class of memory leaks in Android applications. In brief, such a leak occurs when an operating system object of type `Activity` is reachable from a static field after the end of its lifecycle. At this point, the `Activity` object is no

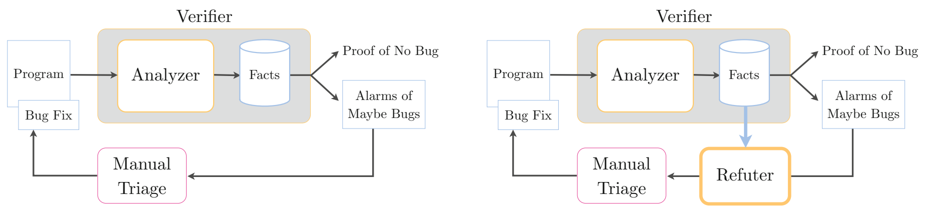

Fig. 1. From verification to alarm triage to bug fixes. The traditional setup (left) requires significant manual effort to triage alarms before bug fixes can be realized. Our approach (right) targets the alarm triage problem with an after-the-fact, automated refutation analysis that soundly filters out as many false alarms as possible to lower the manual triaging effort required. The automated algorithms are outlined in orange, the artifacts that they take as input or output are outlined in blue, and manual effort is outlined in red.

longer used by the operating system but cannot be freed by the garbage collector. For this client, we found the need for a highly precise heap reachability analysis capable of interprocedural path-sensitive reasoning with strong updates. This level of precision is quite difficult to achieve globally in an up-front manner, especially for programs with the size and complexity of our target applications (40K source lines of code plus 1.1M lines of the Android framework).

The conceptual architecture of our approach is diagrammed in Fig. 1. On the left, we show the traditional setup for verification emphasizing that if no proof is obtained, then the report of alarms must be triaged before bug fixes can be produced. On the right, we illustrate that our approach is to drop in a *refuter* stage between the alarm report and the manual triaging process. A refuter is an automated program analysis that tries to prove as many alarms false as possible in a *goal-directed* manner, thereby soundly filtering the list of alarms (hopefully significantly) that manual triaging must consider. For the Android leak client, the "Analyzer" in this architecture is instantiated with an off-the-shelf, state-of-the-art points-to analysis, and the "Refuter" is a tool called THRESHER that implements a highly precise symbolic analysis for refuting heap reachability queries.

In our terminology, a sound refuter is given a query assertion $\widehat{\sigma}$ (i.e., an abstraction of a set of concrete states) at a particular program point ℓ that the up-front analyzer is unable to verify. The goal of the refuter is in essence to derive a contradiction—to prove that there is no possible trace to program point ℓ with any concrete state satisfying $\widehat{\sigma}$. If successful, the alarm is false and can be filtered. Importantly, soundness of refutation analysis can be phrased in a partial correctness sense neither requiring witnessing a concrete trace to a state inside or outside the set abstracted by $\widehat{\sigma}$. In particular, not asking for witness traces enables coarser approximation while being sound with respect to refutations. In our setting, the abstract state $\widehat{\sigma}$ is a separation logic formula constraining a sub-heap, which permits the necessary strong update reasoning, and the analysis is a goal-directed, backwards symbolic analysis that over-approximates all paths to ℓ. The partial-correctness–style soundness criteria enables a relatively simplistic loop invariant inference procedure over separation constraints that turns out to be effective in practice for finding refutations.

While the refuter has the advantage of being goal-directed, the needed level of precision described above is still quite challenging to obtain while simultaneously achieving the required scalability. To achieve better scalability, one contribution of this work is a novel use of the up-front points-to analysis result (diagrammed in Fig. 1 as an arrow from "Facts" to "Refuter"). At a high-level, we enrich the abstract domain in the refutation analysis with a new pure constraint (a `from` constraint) that connects the heap abstraction during refutation analysis with the heap abstraction of the up-front points-to analysis. In essence, the `from` constraint enables reduction [4] with the points-to analysis result to derive contradictions earlier with less case splitting. This reduction is particularly important in our context, as case splitting is highly problematic for backwards analysis with separation constraints (see [3, Sect. 6]).

In the remainder of this extended abstract, we sketch (1) the soundness criteria for refutation analysis and (2) how the `from` constraint enables deriving contradictions earlier with less case splitting. Further details about our approach and methodology are described elsewhere [1, 2]. The bottom line result for our application to Android leak detection is that on a suite of seven Android applications ranging from 2K source lines of code to 40K lines plus the Android framework, THRESHER refuted 172 out of 196 false alarms while exposing 115 true bugs. In other words, THRESHER lowered the false alarm rate from 63% to 17% as compared to the points-to analysis alone—effectively filtering 88% of the false alarms reported by the points-to analysis.

Refutation Soundness

Concrete

$\sigma \in \mathbf{State} \quad s \in \mathbf{Statement} \quad \langle \sigma, s \rangle \Downarrow \sigma'$

Abstract

$\widehat{\sigma} \in \widehat{\mathbf{State}} \quad \gamma : \widehat{\mathbf{State}} \to \wp(\mathbf{State}) \quad \vdash \{\widehat{\sigma}\}\ s\ \{\widehat{\sigma}'\}$

We leave both the notion of concrete state σ and the programming language of interest mostly unspecified. The programming language is assumed only to be an imperative language of statements s defined by a big-step operational semantics judgment form $\langle \sigma, s \rangle \Downarrow \sigma'$ stating that in concrete state σ, evaluating statement s can result in a state σ'. The abstraction $\widehat{\sigma}$ is unspecified except that its meaning is given by a standard concretization function γ. And we write an unspecified abstract semantics for refuting queries as a judgment form deriving a Hoare triple $\vdash \{\widehat{\sigma}\}\ s\ \{\widehat{\sigma}'\}$ stating a pre-condition $\widehat{\sigma}$ and post-condition $\widehat{\sigma}'$ for a statement s.

The soundness condition for refutations in which we are interested can be stated as follows:

Condition 1 (Refutation Soundness)
If $\langle \sigma, s \rangle \Downarrow \sigma'$ *and* $\vdash \{\widehat{\sigma}\}\ s\ \{\widehat{\sigma}'\}$ *such that* $\sigma' \in \gamma(\widehat{\sigma}')$,
then $\sigma \in \gamma(\widehat{\sigma})$.

Informally, given a query post-condition $\widehat{\sigma}'$, we output a query pre-condition $\widehat{\sigma}$ such that if there is some execution of statement s to a concrete state σ' satisfying $\widehat{\sigma}'$, then that execution must begin in a concrete state σ satisfying $\widehat{\sigma}$. Note that the pre- and post-conditions are switched as compared to the standard statement of partial correctness.

This switching in the soundness condition is what we consider defining a goal-directed analysis.

If we derive $\bot$, the abstraction of the empty set of concrete states, for the query pre-condition $\widehat{\sigma}$, then there is no execution of statement s to a state satisfying the query post-condition $\widehat{\sigma}'$. In other words, we have derived a refutation. Our main point is not Condition 1 itself but that witnessing an execution is not required for deriving refutations.

Reducing Separation Constraints with Points-to Facts

The result of a (may) points-to analysis is a points-to graph $\mathring{G}\colon \langle \mathring{V}, \mathring{E} \rangle$. A vertex represents a set of possible memory addresses, typically those allocated at a particular static program point (under some context-sensitivity policy). Such a vertex is typically called an abstract location. An edge from vertex $\mathring{v}_1$ to $\mathring{v}_2$ states that there may be an execution from an address in the concretization of $\mathring{v}_1$ to an address in the concretization of $\mathring{v}_2$. Let us write $\mathring{r}$ for a set of vertices in a points-to graph (i.e., $\mathring{r} \subseteq \mathring{V}$ for a points-to graph $\mathring{G}\colon \langle \mathring{V}, \mathring{E} \rangle$).

In the refutation analysis, let us write $\widehat{a}$ for a symbolic address that abstracts a single concrete address. A from constraint

$$\widehat{a} \;\mathsf{from}\; \mathring{r}$$

relates a symbolic address and a set of abstract locations in the points-to graph in a rather expected way. The meaning is that the concrete address abstracted by $\widehat{a}$ must be in the set of concrete addresses abstracted by $\mathring{r}$ and can be axiomatized as follows for a given points-to graph $\mathring{G}\colon \langle \mathring{V}, \mathring{E} \rangle$:

$$\begin{aligned}
\widehat{a} \;\mathsf{from}\; \emptyset &\iff \mathsf{false} \\
\widehat{a} \;\mathsf{from}\; \mathring{V} &\iff \mathsf{true} \\
\widehat{a} \;\mathsf{from}\; \mathring{r}_1 \wedge \widehat{a} \;\mathsf{from}\; \mathring{r}_2 &\iff \widehat{a} \;\mathsf{from}\; \mathring{r}_1 \cap \mathring{r}_2 \\
\widehat{a} \;\mathsf{from}\; \mathring{r}_1 \vee \widehat{a} \;\mathsf{from}\; \mathring{r}_2 &\iff \widehat{a} \;\mathsf{from}\; \mathring{r}_1 \cup \mathring{r}_2
\end{aligned}$$

For our purposes, if we ever derive $\widehat{a}$ from $\emptyset$, then we have a derived refutation.

To see how from constraints enable earlier contradictions, consider the following example triple:

$$\vdash \left\{ \begin{array}{c} \big(\mathtt{y}\cdot\mathtt{f} \mapsto \mathtt{p} \wedge \mathtt{x} \neq \mathtt{y}\big) \\ \vee \\ \big(\, \mathtt{y} \;\mathsf{from}\; \mathsf{pt}_{\mathring{G}}(\mathtt{x}) \cap \mathsf{pt}_{\mathring{G}}(\mathtt{y}) \wedge \mathtt{x} = \mathtt{y}\big) \end{array} \right\} \quad \mathtt{x.f} := \mathtt{p} \quad \{\, \mathtt{y}\cdot\mathtt{f} \mapsto \mathtt{p} \,\}$$

Here, we write a program variable (e.g., `x`) for the value stored there. The query post-condition $\mathtt{y}\cdot\mathtt{f} \mapsto \mathtt{p}$ states that we are considering a concrete state that has at least one memory location where the contents of field `y.f` is `p`, written $\mathtt{y}\cdot\mathtt{f} \mapsto \mathtt{p}$. In the pre-condition, there are two cases to consider depending on whether or not `x` and `y` alias. In the first disjunct, we are looking for a way to reach the pre-location of the assignment $\mathtt{y}\cdot\mathtt{f} \mapsto \mathtt{p}$ with `x` and `y` not aliasing. In the second, we see if it is possible to reach the pre-location with x and y aliasing, which is a sufficient condition to imply the original query after the assignment. The shaded from constraint intersects the set of possible abstract locations of y with the points-to set of x, written $\mathrm{pt}_{\mathring{G}}(\mathtt{x})$. It says that this case is only possible if the intersection of the points-to set of x and the points-to set of y is non-empty. But even if this intersection is non-empty, the from constraint does not simply "check and forget" but retains whatever restriction obtained by considering the points-to set of x. Intuitively, the $\widehat{a}$ from $\mathring{r}$ constraint abstracts the set of storage locations through which the concrete address corresponding to $\widehat{a}$ flows.

Anecdotally, we have observed that from constraints are particularly important when branching from a method body to all of the call sites of the method (for context-sensitivity). Many of the disjuncts corresponding to the call sites are ruled out by a contradictory from constraint when intersecting with the points-to sets of the actual arguments. Intuitively, when such disjuncts are ruled out, we obtain a goal-directed form of object-sensitivity without the cost of analyzing the code between the argument's allocation site and the call site.

Acknowledgments. This work is joint with Sam Blackshear and Manu Sridharan. We thank the University of Colorado Programming Languages and Verification (CUPLV) group for many insightful discussions. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1055066.

References

1. Blackshear, S., Chang, B.-Y.E., Sankaranarayanan, S., Sridharan, M.: The flow-insensitive precision of Andersen's analysis in practice. In: Yahav, E. (ed.) SAS 2001. LNCS, vol. 6887, pp. 60–76. Springer, Heidelberg (2011)
2. Blackshear, S., Chang, B.Y.E., Sridharan, M.: Thresher: Precise refutations for heap reachability. In: Conference on Programming Language Design and Implementation (PLDI), pp. 275–286 (2013)
3. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM 58(6), 26 (2011)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Symposium on Principles of Programming Languages (POPL), pp. 269–282 (1979)