Finding Cuts in Static Analysis Graphs to Debloat Software

Christoph Blumschein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
christoph.blumschein@student.hpi.uni-potsdam.de

Fabio Niephaus
Oracle Labs
Potsdam, Germany
fabio.niephaus@oracle.com

Codruț Stancu
Oracle Labs
Zurich, Switzerland
codrut.stancu@oracle.com

Christian Wimmer
Oracle Labs
Redwood Shores, USA
christian.wimmer@oracle.com

Jens Lincke
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
jens.lincke@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
robert.hirschfeld@uni-potsdam.de

Abstract

As software projects grow increasingly more complex, debloating gains traction. While static analyses yield a coarse over-approximation of reachable code, approaches based on dynamic execution traces risk program correctness. By allowing the developer to reconsider only a few methods and still achieve a significant reduction in code size, cut-based debloating can minimize the risk. In this paper, we propose the idea of finding small cuts in the rule graphs produced by static analysis. After introducing an analysis with suitable semantics, we discuss how to encode its rules into a directed hypergraph. We then present an algorithm for efficiently finding the most effective single cut in the graph. The execution time of the proposed operations allows for the deployment in interactive tools. Finally, we show that our graph model is able to expose methods worthwhile to reconsider.

CCS Concepts

• **Software and its engineering** → **Automated static analysis**; • **Theory of computation** → *Program analysis*; *Graph algorithms analysis*.

Keywords

Software Debloating, Static Analysis, Call-Graph Construction, Graph Cuts, Interactive Feedback

ACM Reference Format:
Christoph Blumschein, Fabio Niephaus, Codruț Stancu, Christian Wimmer, Jens Lincke, and Robert Hirschfeld. 2024. Finding Cuts in Static Analysis Graphs to Debloat Software. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680306

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680306

1 Introduction

As a software project evolves over time, it tends to accumulate more and more features. The implementation of features that are never used is often called *bloat*. With the growing complexity of software systems, which rely on an increasing amount of dependencies, the amount of bloat shipped tends to rise [25].

This bloat manifests primarily in increased artifact file size, a concern traditionally associated with embedded systems facing memory constraints. Recently, size has gained importance in cloud deployments, too: Secondary effects of software bloat include higher memory footprints, longer build times, an increased attack surface [22], and a reduced optimization potential. Therefore, the field of *debloating*—finding and removing software bloat—has gained attention in recent years [7, 26].

1.1 Context

Many existing solutions deal with the peculiarities of machine code ("binary debloating") [1, 21]. Others operate on the levels of source code or high-level intermediate representations, which facilitate program analysis. The different approaches can be categorized as either *static* or *dynamic*:

Static approaches. These construct some kind of code dependency graph (e.g., a call graph) statically, and identify the unreachable code as bloat. However, static analyses struggle with three difficulties:

a) *Dynamic language features* complicate a sound analysis.
b) Due to over-approximation, they miss much of the bloat.
c) They anticipate all possible program inputs.

The first is usually dealt with by compromising on soundness ("soundiness" [16]), and instead relying on manually provided dependency relationships and information gained from dynamic execution. The second difficulty remains even with soundy analyses, and using more precision only yields diminishing returns. The third one can be addressed by explicitly stating additional assumptions.

Dynamic approaches. Coverage obtained by dynamic execution of the software system under test gives an under-approximation of the necessary code. It can be used in an aggressive debloating strategy [26]. However, the quality of the result heavily depends on the available testcases. Developers employing coverage-based

debloating risk program correctness, unless they can guarantee that every piece of code not covered is indeed dead code. This assurance is impractical for realistic software projects due to their sheer size.

1.2 Cut-Based Approach

We observe that the result produced by static analyses is sufficiently precise for the vast majority of the code, while missed opportunities lead to a strong over-approximation only in a few places. We want to find these critical places so that a developer can consider adding a few assumptions to make the analysis result much more precise. Using our approach, given a small set of deliberately removed (*"cut"*) methods, the static analysis can prove another potentially much larger set of methods as unused. Then, instead of having to supervise the removal of unfeasible amounts of code, the developer only has to consider a handful of locations.

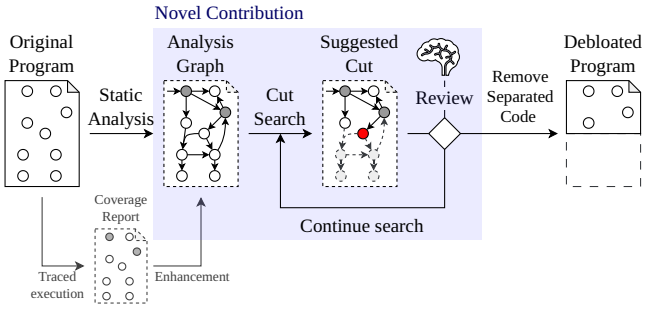


Figure 1: High-level overview.

Proposed Workflow. Figure 1 presents a high-level overview of the proposed workflow. We first capture the rules of an underlying static reachability analysis in a directed hypergraph model. In this rule graph, we search for cuts that separate bloat. The search can be conducted semi-automatically, which allows to target a specific suspicious dependency and to use additional domain knowledge about which code is essential. We also present an algorithm for yielding the cut that separates the biggest chunk of bloat. To maintain the correctness of the program, cut methods must not be exercised in any intended use of the application. Therefore, we restrict the search for cuts to methods not covered during the traced execution of test cases. This restriction alone cannot guarantee correctness, as coverage reports under-approximate the set of necessary code. Therefore, an engineer must review the cut before applying it to the original program.

1.3 Contributions

Our main research question is: *How to find a suitable set of methods for cutting?* As our contributions towards answering the question,

(1) we capture the rules of a static analysis in a directed hypergraph model, which allows for fast recomputation of the analysis given method cuts. This is enabled by introducing
   - an analysis semantics that forms an appropriate trade-off between computation speed and precision,
   - and an algorithm for computing the effect of many disjoint cuts efficiently.
(2) we then develop a tool that uses this model
   - to provide interactive feedback on the amount of bloat removed with a cut,
   - and to recommend effective cuts to the developer.
(3) we evaluate the model's ability to propose effective cuts and show that applying them leads to the expected reduction in code size.

2 Background

In this section, we introduce the necessary theory, and establish some formalism which we use throughout the paper.

2.1 Directed Hypergraphs

Directed hypergraphs generalize directed graphs by allowing edges to connect arbitrarily large sets of nodes. In this paper, we only deal with directed B-hypergraphs [10]. In a directed B-hypergraph, the edges have a set of source nodes (*tail*), and exactly one destination node (*head*). Formally, we define a directed hypergraph $\mathcal{H} = (V, E)$ as containing nodes V and hyperedges $E \subseteq 2^V \times V$. We define two helper functions for denoting the *head* and *tail* of a hyperedge:

$$T : E \to 2^V, \quad H : E \to V \quad \text{such that} \quad e = (T(e), H(e)) \quad \forall e \in E$$

A hyperpath (also called B-hyperpath) from a set of nodes S to a single node t is a sequence of edges $e_1, ..., e_n$ such that

$$T(e_i) \subseteq S \cup \{H(e_j) \mid 1 \leq j < i\} \quad \forall i = 1, ..., n$$

We write $S \to^*_H t$ if there is such a path. Notice that the start set of nodes S can be encoded in the graph via "entry" hyperedges with an empty tail, such that

$$S \to^*_{(V,E)} t \iff \emptyset \to^*_{(V,E')} t \quad \text{with} \quad E' = E \cup \{(\emptyset, s) \mid s \in S\}$$

As the graphs in this paper follow this encoding, we are commonly interested in the set of nodes reachable from entry hyperedges:

$$\text{Reach}(\mathcal{H}) = \{t \in V \mid \emptyset \to^*_{\mathcal{H}} t\}$$

We say "t is reachable in $\mathcal{H}$" if $t \in \text{Reach}(\mathcal{H})$.

Traversal. The set of reachable nodes can be computed in linear time using a simple worklist algorithm. Finding a minimum-weight path to a particular node is sometimes feasible, depending on the weight measure [10]. Minimizing the cardinality of a hyperpath is NP-hard. Yet, for inductively defined weight measures, there often exist efficient algorithms. One example for that is *Rank*, which assigns each node along the path the maximum value of its predecessors plus one. Minimum-*Rank* hyperpaths can be computed via a generalized Breadth-First Search (BFS).

Cuts. We say "$X \subseteq V$ is a cut for $t \in V$ in $\mathcal{H}$" iff $t \notin \text{Reach}(\mathcal{H} - X)$. The problem of finding a minimum cardinality cut is equivalent to Maximum Horn Satisfiability, which makes it NP-hard [2] as well.

2.2 Call Graph Construction

Much research in the static analysis field deals with the construction of call graphs. A call graph is a simple directed graph, containing edges from each method to any potential callee. It may contain spurious edges; in fact, all practical analyses produce over-approximations of the "true" call relationship.

A big problem is posed by dynamic languages features, such as reflection (i.e., string-based dispatch) [14]. In order to produce useful results, static analyses usually compromise on soundness when encountering reflection, and rely on a combination of user-provided rules and heuristics for simple cases [5].

Various call graph construction algorithms differ in how they treat multiple dispatch in object-oriented languages, which corresponds to first-class function invocation in functional languages [19]. In the following, we present some construction algorithms, beginning with a very coarse approach and gradually advancing towards more precision.

Reachability Analysis. The simplest analysis ignores run-time polymorphism. Caller-callee pairs are matched by the name of the called method.

Class Hierarchy Analysis. A first improvement is to take the class hierarchy into account [8]. Unrelated methods that happen to have the same name are correctly excluded. Also, upon a run-time polymorphic call, only implementations belonging to a type assignable to the declared type of the call are connected via an edge.

Rapid Type Analysis (RTA). RTA considers callees only if their declared type (or a type inheriting the same implementation) has previously been instantiated in a reachable method [3].

Variable Type Analysis (VTA). VTA tracks the flow of instantiated types along variable assignments from allocation sites to invocations. It keeps track of the set of types that may appear in each local variable or field. A callee is only considered if the receiver variable may contain its defining type.

In the initial VTA paper [28], the authors avoid "on-the-fly call-graph construction" [19], which would be required to obtain the minimum fixed-point universe. Instead, after initializing their call graph with an approximation computed by a simpler, less precise analysis, they propagate type sets in one run along the topological order of Strongly-Connected Componentss (SCCs), and remove edges where the receiver variable does not contain any defining type of the callee implementation. Therefore, they trade precision for reduced computational effort.

Nevertheless, whenever we use the term VTA in this paper, we refer to a minimum fixed-point variant. So only once a new callee is reachable, types are propagated from actual to declared parameters, and from declared to actual returns. In the Points-to Analysis (PTA) community, this is also known as Context Insensitive Control Flow Analysis (0-CFA) [29].

3 Capturing an Analysis Run in a Directed Hypergraph

Given a static analysis and a software project, we want to find promising cuts. Instead of performing the analysis over and over again, which can be time-consuming, we capture one run in a graph. In principle, any analysis that derives facts via rules (horn clauses) is suitable for our approach. This includes the many points-to analyses that can be specified as a high-level datalog program [24].

We derive the rule graph by mapping every fact to a node and every rule instantiation to an edge. The graph is ordinary iff all rule preconditions consist of at most one atom. This is only the case for the most simple analyses, such as the Reachability Analysis and CHA described in Section 2.2. In general, however, we obtain a directed hypergraph.

Generally, a higher level of analysis precision results in a larger graph. While RTA as an underlying analysis yields a compact hypergraph, VTA already results in an explosion of the graph, as every combination of types and variables (beyond 10^{10} in practice) yields a distinct node. Therefore, we introduce an approximation of VTA, which we denote VTA$^{<k}$. This *saturation* technique models the exact types that may occur in each variable as long as the number of types stays below a certain threshold and merges variables that exceed this threshold [32]. This enables us to limit the run time until a BFS converges.

3.1 Rules

The following subsection presents the rules of VTA$^{<k}$. In addition to the precondition atoms, which are stated on the upper side of each rule, a rule may have a meta-condition written on the right. These introduce information about the concrete program, which is assumed to stay constant during cutting and re-simulating.

Methods. The relation $\textsc{MethodReachable}(\cdot)$ keeps track of the methods that are required according to the analysis. As the analysis begins at predefined entry points (e.g., the main method), these are unconditionally reachable.

$$\text{Entry}\ \frac{}{\textsc{MethodReachable}(m)}\ \text{if } m \text{ is an entry point}$$

Direct calls—when the callee can be resolved statically—can be modeled with a simple rule.

$$\text{Call}\ \frac{\textsc{MethodReachable}(m_1)}{\textsc{MethodReachable}(m_2)}\ \text{if } m_1 \text{ directly calls } m_2$$

Variables. The relation $\textsc{Flow}(t, v)$ tracks which type t may appear in a variable v.

$$\text{Alloc}\ \frac{}{\textsc{Flow}(t, v)}\ \text{for "}\texttt{v = new t(...)}\text{"}$$

Variables propagate their types along assignments, whether explicitly stated in the code or implicitly established through method parameter/return linking. Each variable applies a filter on incoming types. It is derived from explicit casts, control-flow facts, and the implicit casting of the receiver argument at run-time-dispatched method implementations. Variables defined in method bodies only propagate their types once the method is reachable since propagation can have side effects on the reachable universe.

$$\text{Propagation}\ \frac{\textsc{Flow}(t, v_1) \quad \textsc{MethodReachable}(m)}{\textsc{Flow}(t, v_2)}\ \text{for "}\texttt{v2 = v1}\text{" if } \bigwedge_{m \text{ defines } v_1}^{t \in \text{filter}_{v_2}}$$

For variables defined outside of methods (i.e., fields), there is an analog rule that omits the $\textsc{MethodReachable}(m)$ precondition.

A polymorphic implementation method becomes reachable once its formal receiver ("this-pointer") contains any type. This works as intended since the formal receiver variable applies the correct

filtering, accepting only types that use this exact implementation—subtypes that have not overridden the method themselves.

$$\text{Invocation}\ \frac{\textsc{Flow}(t, v)}{\textsc{MethodReachable}(m)}\ \text{if } v \text{ is formal receiver of } m$$

Type Set Merging. To restrict the number of types whose propagation has to be computed for each variable, we employ an approximation: once a variable may contain k different types, we consider it saturated.

$$\text{Saturation}\ \frac{\textsc{Flow}(t_1, v) \quad \ldots \quad \textsc{Flow}(t_k, v)}{\textsc{Saturated}(v)}\ \text{with } t_i \neq t_j \ \ \forall i \neq j$$

Then, instead of tracking its precise type set, we only track one globally merged set of types for all saturated variables.

$$\text{Merge I}\ \frac{\textsc{Saturated}(v) \quad \textsc{Flow}(t, v)}{\textsc{Merged}(t)}$$

$$\text{Merge II}\ \frac{\textsc{Saturated}(v) \quad \textsc{Merged}(t)}{\textsc{Flow}(t, v)}$$

With a higher limit k, $\text{VTA}^{<k}$ becomes more precise. Observe that RTA and VTA correspond to special cases of it:

$$\text{RTA} = \text{VTA}^{<0} \underset{\text{precision}}{<} \ldots \underset{\text{precision}}{<} \text{VTA}^{<\infty} = \text{VTA}.$$

3.2 Graph Model

Mapping each $\textsc{Flow}(\cdot, \cdot)$ fact to a separate node would result in a graph size explosion. Instead, taking advantage of the parallel structure of the nodes induced by *Propagation*, we introduce special *variable* nodes, which represent a vector of $\textsc{Flow}(\cdot, \cdot)$ facts. The vectors are of size $|Types|$, as they represent a subset of all types in the analysis universe. Furthermore, we integrate saturation semantics into these nodes to efficiently evaluate the cardinality constraint in the *Saturation* rule.

Figure 2 illustrates the behavior of the nodes in our graph in terms of a circuit diagram. There are two kinds of nodes in our graph: ordinary nodes modeling methods and special nodes for variables. In a BFS, the ordinary nodes behave unsurprisingly: each one stores a state of one bit, indicating whether it is reachable. Once any predecessor becomes reachable, an ordinary node becomes reachable, making all its successors reachable. A variable node stores a vector of bits, indicating whether each type can appear in this variable. It is interconnected to other variable nodes according to assignments in the program. Each variable node has a read-only parameter $filter_v$, which masks the types that may be propagated from predecessors. Once its state includes at least k types, it is united with the global set $\textsc{Merged}(\cdot)$. If a variable is not a field, its node is associated with the node of the method in which it is defined. The reachability of the method node acts as a gate to the further propagation of the types. If the variable is the formal receiver of its method, it leads to the reachability of the method node once any type occurs in its state.

Strictly speaking, our graph with special nodes is no longer a directed hypergraph. Yet, since it compactly represents something that could be reduced to a hypergraph in polynomial space, theoretical results from hypergraph theory apply. Furthermore, given an implementation of a BFS algorithm that respects the particular

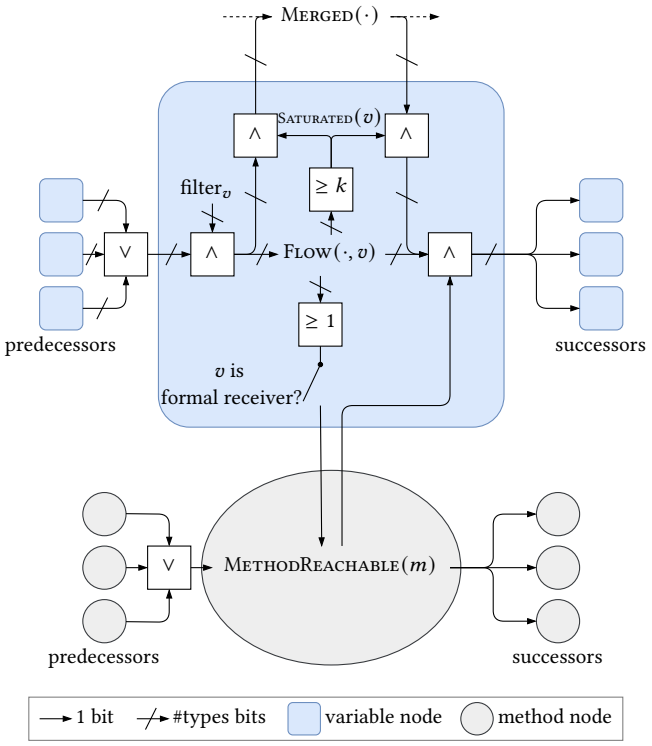


Figure 2: Node semantics expressed as a circuit diagram.

semantics of variable nodes, we can treat our special graph like a directed hypergraph in the following sections.

4 Analysis Recomputation with Cuts

As the analysis discovers facts monotonically, removing a set of nodes X and subsequently doing a graph search yields all elements that the analysis would discover if the corresponding code elements had not existed. Compared with re-running the analysis, "re-simulating" it with a given cutset speeds up the hypothesize-modify-observe cycle of a developer trying to reduce the size of a software artifact. In addition, this basic operation facilitates the construction of automated searches on top of it.

4.1 Finding Promising Cutsets

We use the function *Sep* to denote the set of nodes separated by a given cut.

$$\text{Sep}_G : 2^{V(G)} \to 2^{V(G)}$$

$$\text{Sep}_G(X) = \text{Reach}(G) \setminus \text{Reach}(G - X)$$

Depending on the scenario, a *promising* cutset is a solution to one of multiple optimization problems. If developers suspect that a particular dependency is bloat, they might want to find a small set of methods that separate it.

$$\text{MinCut}_G(T) = \underset{\substack{X \subseteq M \\ T \subseteq \text{Sep}_G(X)}}{\text{argmin}} |X|$$

Without specific targets in mind, a developer may be interested in some methods that reduce the artifact size as much as possible.

$$\text{MostEffectiveCut}_G(l) = \operatorname*{argmax}_{\substack{X \subseteq M \\ |X| \leq l}} \sum_{m \in \text{Sep}_G(X)} \text{size}(m)$$

Here, $M \subseteq V(G)$ denotes the set of methods we allow to cut. This additional constraint can be used to restrict the result to methods not included in dynamic coverage or to incorporate other project-specific knowledge of a developer.

While *MinCut* would be computationally feasible if we dealt with a regular graph, it is NP-hard for hypergraphs (see Section 2.1). *MostEffectiveCut* is a variant of Minimum-Size Bounded-Capacity Cut (MinSBCC) [11], which is already NP-hard for regular graphs. Therefore, we focus on $\text{MostEffectiveCut}_G(1)$, the singleton cut problem.

4.2 Simulating Many Disjoint Cuts

The naïve approach to determining $\text{MostEffectiveCut}_G(1)$ involves computing $\text{Sep}_G(\{m\})$ from scratch for each method $m \in M$. This is impractical for larger projects with more than 100,000 methods. This section describes a divide-and-conquer algorithm that reuses intermediate results to compute $\text{MostEffectiveCut}_G(1)$.

Incremental Computation. It would be convenient if we could compute $\text{Reach}(G)$ once, and take a decremental step to $\text{Reach}(G - \{m\})$ for each method m with little effort. Updating the set of reachable elements after deletion is difficult to do efficiently. However, updating the set of reachable elements after an increment is easily computed in time linear to the increase in reachable nodes $\Delta = |\text{Sep}_G(X)|$. We exploit this by approaching the desired reachability results from the other side in a hierarchical scheme.

Let $M = \{m_0, ..., m_{n-1}\}$, with n being a power of two for the sake of simplicity. Then we define hierarchical partitions:

$$X_w^i = \left\{ m_j \in M \;\middle|\; \frac{n}{w} \cdot i \leq j < \frac{n}{w} \cdot (i+1) \right\}$$

The definition above implies recursive properties:

$$\begin{aligned} X_1^0 &= M \\ X_w^i &= X_{2w}^{2i} \;\dot\cup\; X_{2w}^{2i+1} && \text{for } 0 \leq i < w \leq \frac{n}{2} \\ X_n^i &= \{m_i\} && \text{for } 0 \leq i < n \end{aligned}$$

Using hierarchical partitioning, Algorithm 1 efficiently enumerates all universes with a single cut. It starts with computing $\text{Reach}(G - M) = \text{Reach}(G - X_1^0)$, the reachable nodes in a graph where all allowed methods are removed. In each recursive step, given $\text{Reach}(G - X_w^i)$, it computes $\text{Reach}(G - X_{2w}^{2i})$ and $\text{Reach}(G - X_{2w}^{2i+1})$ respectively, by adding the complementary set of nodes to the graph and continuing the search (see Figure 3).

Complexity. The theoretical worst-case run-time $O(n^2)$ remains the same as for the naïve approach, since large parts of the graph could be only discovered late in the computation tree. We expect benign graphs to satisfy $|\text{Reach}(G)| - |\text{Reach}(G - X)| \in O(|X|)$ in the average case. Then, at each layer of the tree, $O(n)$ increases in the reachability relation are accumulated. As the tree is balanced, it has $O(\log(n))$ layers. Therefore, we expect the average-case complexity of this algorithm to be $O(n \cdot \log(n))$. Section 6 shows that this algorithm indeed performs better by orders of magnitude. An additional speedup can be gained by applying a Branch-and-Bound strategy.

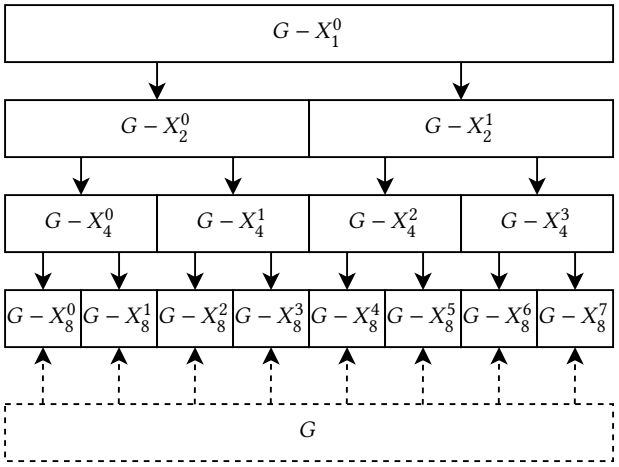


Figure 3: Incremental scheme: Updating the set of reachable nodes after a decremental change is not computationally feasible. Instead, we start with many cuts and incrementally continue the graph search after adding methods back again in a hierarchical manner. This allows us to efficiently enumerate the reachable subgraphs for all possible single-method cuts.

Algorithm 1 Enumerate $\text{Reach}(G - \{m\})$ for all $m \in M$

Require: G is a hypergraph, $M \subseteq V(G)$
  $R \leftarrow \text{Reach}(G - M)$
  RECURSIVE(G, M, R)

  function RECURSIVE(G, X, R)
    if $X = \{m\}$ **then**
      yield (m, R)
    else
      $X_0, X_1 \leftarrow \text{split}(X)$
      for $i \in \{0, 1\}$ **do**
        $j \leftarrow 1 - i$
        $R' \leftarrow$ INCREMENTALSEARCH(G, X_i, X_j, R)
        RECURSIVE(G, X_i, R')

  function INCREMENTALSEARCH(G, X, Y, R)
    Expects $R = \text{Reach}(G - X - Y)$
    Returns $\text{Reach}(G - X)$

5 Implementation

In our implementation, we capture the static analysis results of GraalVM Native Image, which produces standalone binaries for Java applications that contain the application along with all its dependencies [31]. GraalVM Native Image uses a static analysis that determines a reachable universe to limit the amount of code that needs to be compiled and shipped. We obtain our rule graph via an instrumentation of its imperative analysis code.

5.1 Extended Rule Graph

In principle, the graph generated by our concrete implementation still follows the same structure as described in Section 3.2. We extended it to deal with issues that typically occur when analyzing real-world Java projects.

Additional Facts. In addition to MethodReachable(·), we have ordinary graph nodes that model more kinds of facts. These include class and field reachability, types instantiated during build-time, the effects of custom analysis plugins, and statically resolvable reflection usages.

Dynamic Language Features. Languages like Java provide certain dynamic features which make sound analysis infeasible. GraalVM Native Image already handles them with a combination of static resolution in trivial cases, tracing concrete executions, and user-defined configuration. We introduce the respective facts and add edges between them, where applicable. These facts might be considered entry points, e.g., if the user-defined configuration specifies that a method should be accessible per reflection. They can also be integrated via incoming edges, e.g., if the analysis resolves a trivial reflection usage statically. The latter scenario is preferable, as it captures dependencies more precisely, increasing our model's capability to propose effective cuts.

Build-Time Initialization. In addition to Ahead-of-time (AOT)-compilation, Native Image reduces startup time by executing class initializers at build-time, if possible. The result of such early execution manifests in a build-time heap, which is persisted as part of the resulting binary and, therefore, subject to the analysis, too. To make these heap effects fit into our framework of facts and causal rules, we employ a Java Virtual Machine Tool Interface (JVMTI) agent. It tracks heap allocations and write operations during build time and assigns the effects observed by the analysis to the responsible class initializers.

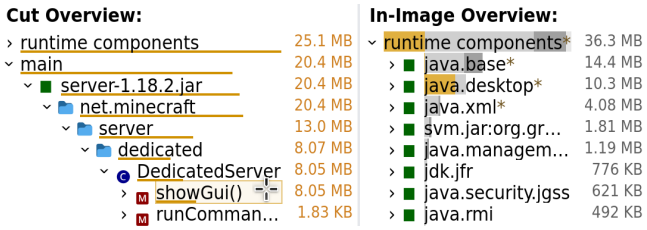
5.2 Cut Tool

While our graph model allows us to compute the effects of cuts efficiently, the *Cut Tool* serves as the front end, enabling the developer to interact with the model. It is implemented as a client-side-only web application. The heavy lifting part, i.e., recomputing the analysis as described in Section 4, is done in a hand-written library shipped to the client as a WebAssembly module.

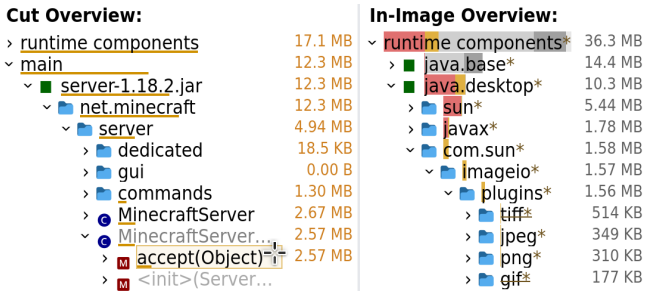
We demonstrate how the Cut Tool can be used for debloating using the Minecraft Server[1] as a concrete real-world example. It is used to host a multiplayer session for a game written in Java. Our prior knowledge of the software is limited as we did not create it.

First, we played the game to generate coverage information. It is not exhaustive, but it should be sufficient to prevent the Cut Tool from suggesting overly aggressive cuts. After supplying the coverage-enhanced rule graph to the Cut Tool, we see a user interface mainly composed of two views: They both list the code elements in their natural hierarchy, similar to the project overview in an IDE. Elements of the hierarchy are modules, packages, classes, and methods. The left panel offers elements that are available for cutting. The right panel shows what is separated by the selected cut

[1] https://www.minecraft.net/de-de/download/server



(a) Cutting `DedicatedServer.showGui()` separates 8.05 MB.



(b) Additionally cutting `MinecraftServer$$Lambda.accept()` separates 2.57 MB.

Figure 4: The effect of cuts indicated in our Cut Tool.

set. Each element comes with a bar of a certain length. On the right side, the bar of an element indicates the code size of all methods in its subtree. On the left side, it indicates the code size of all methods separated after cutting the element.

Cut View. The batched cut simulation scheme introduced in Section 4.2 allows us to efficiently compute the cut effectiveness for all expanded elements on the left panel. By sorting them accordingly, a developer can explore the ones that are most worth reconsidering. When hovering over an element on the left side, the bars on the right side are partially filled in orange to show a breakdown of code size reduction across the hierarchy (see Figure 4). Clicking on an element on the left side can add it to the cut set. Then, the code size reduction previewed on the right side in orange turns red permanently. Additionally, the cut effectiveness is recalculated for each element on the left side. It now considers how much *additional* code would be removed if the element was added to the cut set. The cut effectiveness often decreases, indicating that additionally cutting an element is redundant given the current cut selection. Yet, it can also increase if a synergy effect arises between the existing cut selection and the potential new cut.

The Cut Tool also provides a button to compute the most effective cut method. Conceptually, it automates the process of expanding every element to find the method with the largest bar. Applying it twice for the Minecraft Server yields two cut methods:

(1) `DedicatedServer.showGui()`
(2) `MinecraftServer$$Lambda.accept()`

Cut 1 separates 8.05 MB code in the module `java.desktop` (see Figure 4a), while cut 2 separates an additional 2.57 MB (see Figure 4b). Without the first cut, the second cut would only separate 0.56 MB.

Figure 5: Hyperpath to `java.awt.image.ComponentColorModel`.

So there must be code in `java.desktop` that becomes reachable through either one of these two methods.

Reaching Path View. If developers suspect that an element is bloat, they can obtain an explanation for why it has been considered reachable in the form of a hyperpath. It is rendered as a Directed Acyclic Graph (DAG) from bottom to top, with the target as the topmost element. The developer may then either figure out that the suspicious element is needed after all or find out where to start cutting. As there can be multiple reaching hyperpaths, cutting an element from the currently displayed path does not necessarily make the target unreachable. In this case, the view is updated to show another path.

It can be assumed that smaller hyperpaths are easier to understand. However, the problem of finding a path with the smallest number of nodes is NP-hard (see Section 2.1). In our implementation, we present a path with minimum rank instead. The solution happens to be benign in many cases. However, in a big project, after cutting through paths of manageable size, we sometimes encounter hyperpaths spanning more than 1,000 nodes.

Figure 5 shows why `ComponentColorModel`, a class in `java.desktop`, is still reachable after our first cut: The second cut method, passed as a lambda argument to `Optional.ifPresent()` by `MinecraftServer.updateStatusIcon()`, uses `ImageIO`.

Bloat Identification. The source code shows that the first cut method is invoked if a command line option requests the Graphical User Interface (GUI). As the GUI provides little benefit apart from wrapping a command line in a custom window, it is typically disabled and also was not exercised in our coverage run. The second cut method is responsible for sending a customizable server icon to clients. If an icon PNG is in the current directory of the server process, it goes through a questionable round trip via `ImageIO.read()` and `ImageIO.write()` before being sent again as a PNG. When dealing with the `ImageIO`-API, the static analysis could not reason about potentially encountered file formats. Therefore, the program is bloated with code for encoding and decoding various formats, such as TIFF, JPEG, PNG, GIF, and BMP.

The application can be successfully debloated by replacing the two cut methods with exception-throwing stubs and compiling again with GraalVM Native Image. However, the debloated program fails with an exception if the command line arguments request the GUI. Therefore, the user of our approach is responsible for ensuring that cut methods are never exercised in the anticipated usage scenario.

6 Evaluation

Throughout this section, we compare the rule graph capturing a VTA$^{<20}$ against one coarser approximation capturing a RTA to find out whether the additional effort is worthwhile. The RTA graph omits variable nodes and instead uses hyperedges to directly connect virtual invocations and type instantiations with overridden methods, effectively circumventing the precise tracking of types through variables. Note that we denote the graph model with variable information as VTA (without saturation cutoff). We do that because saturation is only applied when querying the rule graph.

6.1 Projects

We use the following applications for the evaluation:

- *Console HelloWorld*: A simple Java application printing a text to the standard output.
- *DaCapo, Renaissance*: Benchmark projects that are compatible with GraalVM Native Image.
- *{Quarkus, Micronaut, Spring} HelloWorld*: Simple applications for popular Java frameworks [18, 20, 30].
- *Quarkus Tika*[2]: A Quarkus application for parsing documents using the Apache Tika library[3].
- *Micronaut ShopCart*: A demo application for the Micronaut framework.
- *Spring PetClinic*[4]: A large demo application for the Spring framework.

Table 1 gives an overview of the size of each project: It shows the number of methods discovered by the analysis of GraalVM Native Image and the number of nodes in our graph model. The number of nodes is reported for the VTA graph. The RTA graph has approximately the same number of simple nodes but omits the variable nodes.

6.2 Graph Collection Performance

Before the graph can be used for analysis recomputation, it has to be collected once. For this, we instrumented the static analysis of Native Image, causing some overhead. While the performance of the collection is not critical to our approach, the overhead should not exceed orders of magnitude.

For benchmarking the data collection, we use Spring PetClinic, as it is the largest project we consider. Figure 6 shows the run-time

[2] https://github.com/quarkiverse/quarkus-tika
[3] https://tika.apache.org/
[4] https://github.com/spring-projects/spring-petclinic

Table 1: Analysis recomputation times for selected projects using either RTA or VTA$^{<20}$.

	Project size (in thousands)			Time per Operation (ms)					
	Reachable	Graph Nodes		Single Cut		Batched Cuts		Most Effective Cut	
Project	Methods	Ordinary	Variable	RTA	VTA$^{<20}$	RTA	VTA$^{<20}$	RTA	VTA$^{<20}$
Console HelloWorld	17	99	56	6	23	0.001	0.006	21	109
Renaissance: future-genetic	26	154	93	12	46	0.002	0.011	51	232
DaCapo: luindex	27	170	107	13	47	0.002	0.009	51	226
Quarkus HelloWorld	53	247	239	25	106	0.007	0.054	125	654
Micronaut HelloWorld	67	421	315	38	150	0.009	0.069	243	1,183
Spring HelloWorld	80	528	400	46	182	0.004	0.032	292	1,310
Quarkus Tika	83	390	401	42	184	0.008	0.059	289	1,406
Micronaut ShopCart	74	465	350	44	179	0.008	0.070	263	1,337
Spring PetClinic	195	1,278	1,055	120	654	0.004	0.070	747	5,309

of GraalVM's static analysis with different collection options: *none*, *RTA*, *VTA*, and *VTA with JVMTI agent*. Disabling the JVMTI agent is possible but forces the graph model to over-approximate untracked effects by introducing more entry edges (see Section 5.1).

Our benchmark machine has 16 logical cores. We ran each configuration multiple times and observed a negligible standard deviation. We measured the elapsed wall-clock time as well as the CPU-time across all threads. Since they did not grow proportionally, we analyzed the parallel scalability: assuming that the program consists of a perfectly parallelizable and a sequential part, according to Amdahl's law, we derived the parallel and sequential part of the wall-clock execution time.

The results show that collecting the RTA model introduces an overhead of 43%. Collecting the VTA model already needs significantly more time, accumulating to +191%. However, we see that only the sequential part increased significantly. The post-processing of the variable assignment graph is executed in a single thread. As we have not spent any effort on its parallelization, this is currently a bottleneck. Having the JVMTI agent hooked into the build process results in a greater increase in build time, leading to an overhead of up to 7x. So, while the current implementation has some parts that introduce high costs, capturing the rule graph via instrumentation of the analysis code itself is feasible.

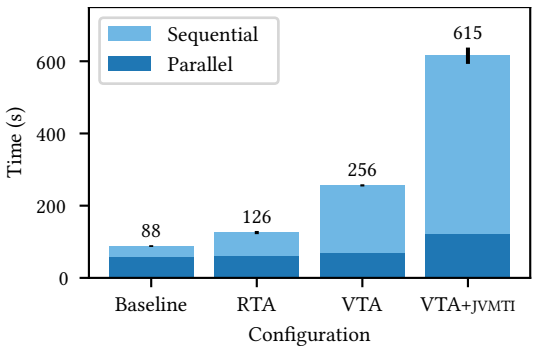


Figure 6: Analysis data collection times for Spring PetClinic.

6.3 Analysis Recomputation Performance

We evaluate the performance of three key operations that are defined on our model:

Single Cut. The run-time of a single cut simulation depends on the actual cut set. As the operation is implemented using a BFS with a subset of nodes marked as forbidden, we obtain the worst-case run-time with an empty cutset.

Batched Cuts. We evaluate our incremental algorithm by cutting a batch of (up to) 50,000 randomly selected nodes in succession. Then, we report the average time needed for a single cut result.

Most Effective Cut. We use the incremental algorithm to compute $\text{MostEffectiveCut}_G(1)$ ten times in succession. The difference between this and the evaluation of *Batched Cuts* is twofold: On the one hand, we now consider all methods except the ones included in coverage, which is generally a higher number. On the other hand, employing a Branch-and-Bound strategy allows us to skip the enumeration of most cut results. The computation of the n-th suggested cut does not correlate with n significantly. Yet, since bounding can happen earlier or later, there is some variance in the time needed for the next cut. Therefore, we report the average time needed for each of the ten cut suggestions.

Results. All benchmarks are single-threaded and were run on an Intel i7 4790 at 3.6 GHz, with Turbo Boost disabled. The results are shown in Table 1.

In general, we see that the RTA approximation is consistently faster, yielding a speedup factor ranging between 3-9. Yet, even for the large PetClinic, the time needed for the analysis recomputation allows for an interactive setup with both configurations [12]. Furthermore, we see that our divide-and-conquer algorithm is indeed much faster on real-world inputs despite having the same worst-case time complexity as simulating single cuts repeatedly. Finding the most effective cut only takes up to a few seconds, even with the PetClinic using VTA$^{<20}$.

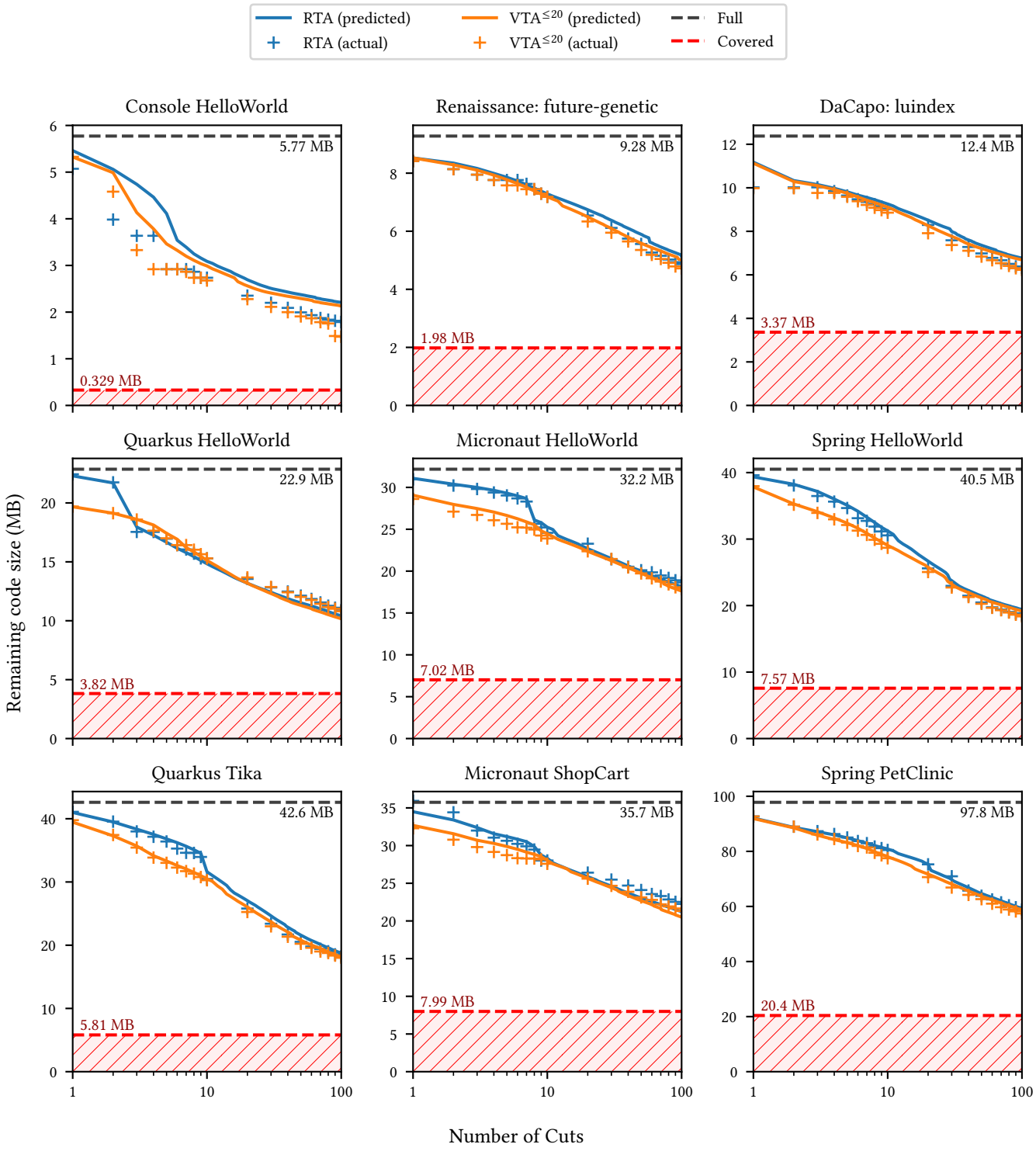


Figure 7: Effectiveness of the first 100 suggested cuts for various projects.

6.4 Effectiveness of Suggested Cuts

The central assumption of this paper is that the dependency information contained in the graph is useful for finding effective cuts. In order to test that assumption automatically, we repeatedly searched for and applied the most effective cut 100 times. For a developer, a cut set of size 100 should already be considered too large to handle. Yet, we are curious to see how the cut effectiveness develops as the number of cuts increases.

To avoid getting suggested unrealistic cuts, we again restricted the set of methods available for cutting with coverage information. The coverage information was obtained through manual testing or by running each benchmark, depending on the subject application type. Additionally, we compiled the projects with the suggested cut set of methods removed to validate the prediction made by our graph.

In Figure 7, we plotted the remaining code size of a project against the number of methods that were cut. The continuous lines show the predicted debloating effect, while the crosses show the actual result after compilation with GraalVM Native Image. The black horizontal line indicates the size without cuts. The red horizontal line shows the size of all methods included in the coverage report, setting a hard boundary on how much can be separated. We chose a logarithmic axis for the number of cuts, as we observe diminishing returns: usually, with more cuts, each additional cut is less effective than the previous ones. However, there are exceptions. In cases where multiple cuts are required to eliminate some part of the universe, an additional cut can lead to a greater reduction in size.

Results. First, we see that our model accurately predicts the actual debloating effect in general. An exception is the *Console HelloWorld* project: Our model overestimates the remaining code size. We attribute this to the very small coverage, allowing our algorithm to suggest cutting core JDK methods, some of which get special treatment in GraalVM Native Image.

In all projects, the advantage of $\text{VTA}^{<20}$ becomes small when applying 100 cuts. For fewer cuts, the situation varies: The *Renaissance*/*DaCapo* benchmark projects show only a small difference between RTA and $\text{VTA}^{<20}$. In the framework projects, $\text{VTA}^{<20}$ starts with a distinct advantage for a small cut set. In the Quarkus- and Micronaut-based projects, RTA suddenly catches up at around 10 cuts. Oddly, in the *Quarkus HelloWorld* project, RTA suddenly yields a better result than $\text{VTA}^{<20}$: With exactly three cuts, the greedy strategy happens to select methods that form a globally more optimal combination.

Overall, we see that while the advantage of $\text{VTA}^{<20}$ over RTA is not proportional to the increase in computation time needed, it can yield significantly more effective suggestions, especially if we demand smaller cutsets.

7 Limitations

Decrementality. Our graph only contains those rule instantiations and facts discovered by the captured analysis, so it can only predict the effects of code removal. If a developer considers replacing the usage of a particular API with another, our model cannot predict the potential benefit. Additionally, when fitting an imperative static analysis into our declarative rule-based model, the soundness of predictions relies on the analysis code behaving monotonically.

Precision. While sophisticated PTAs could be captured and recomputed with our approach, the high number of discovered facts leads to an extended run-time, prohibiting interactive use. Furthermore, in our concrete model, only the removal of entire methods can be modeled due to the graph's granularity. If predictions on a finer granularity level are desired, e.g., for removing individual lines, more nodes would need to be created in the graph.

8 Related Work

This section presents related work on debloating, understandability of static analyses, and points-to analyses.

8.1 Debloating

The problem of debloating Java applications has been approached with multiple strategies. *JShrink* [7] uses static analysis to compute the set of reachable methods. For dealing with dynamic language features, it relies on execution traces.

Coverage-Based Debloating. The authors of "Coverage-Based Debloating for Java Bytecode" [26] show that *JShrink* yields universes significantly larger than the code covered by automated tests and, therefore, still contain potential bloat. We observe a similar situation with our projects built with GraalVM Native Image, which include more methods than those actually exercised at run-time. The authors then applied coverage-based debloating to a large set of libraries, which reduces artifact size to a minimum. While the large majority of the client projects' individual tests still passed when using the debloated libraries, many clients had at least one broken test. So ultimately, they cannot guarantee that the test suite covers the behavior of actual programs completely.

Input Specialization. Approaches such as TRIMMER [22] start with a user-provided manifest constraining the environment at execution time. This can include the contents of program arguments, environment variables, and configuration files. A program transformation then uses partial evaluation techniques to specialize the program, eliminating function calls that belong to certain unused features. Similar to our approach, TRIMMER eliminates code that can be proved dead after adding assumptions. The domain of its assumptions differs from ours: it assumes input strings to be known constants at compile-time, while we assume our cut methods to be unreachable. Unlike our approach, TRIMMER can only be used for prediction. It cannot suggest input constraints suitable for eliminating specific features or removing a large amount of code.

8.2 Understanding Static Analyses

With more sophisticated analyses employed, slight changes to the code can greatly affect the outcome in a desirable way. Therefore, the question of how to convey the verdict of static analysis to the developer is increasingly getting attention.

Optimization Coaching. Optimization Coaching aims at informing developers about missed opportunities to optimize their program [27]. The authors introduce a textual feedback channel from the optimizing compiler to the developer. For example, when declaring a variable with a more concrete type would allow the optimizer to eliminate virtual dispatch, the developer receives a suggestion to

reconsider the typing decision. To implement this, the authors annotate the optimizer's code patterns with log messages that are emitted when the preconditions of an optimization are nearly missed. Their concept of a near miss is similar to a small but effective cut set in our rule graph. However, their system cannot reason transitively, as further optimizations only enabled by a nearly missed one are not considered.

Explaining Static Analyses with Rule Graphs. Rule graphs have been introduced to explain the outcome of a static analysis to the user [9]. The authors report an improved ability of users to understand analysis warnings and identify weak analysis rules. Their running example is a taint analysis that reports false positives. Similar to our graph model, the insights gained should enable users to modify the program so that the analysis verdict matches the actual behavior more closely. Also, it can be used as a feedback mechanism to the analysis developer.

8.3 Points-to Analyses

PTAs track allocated objects throughout a program. They differ primarily in the granularity of the equivalence classes used for modeling allocated objects and variables ("contexts") [23]. Analyses with k-caller context-sensitivity maintain many different sets for a variable, distinguished by which k methods precede on the abstract call stack. When such a PTA is used to construct a call graph, it is also called *k*-context sensitive Control Flow Analysis (k-CFA).

Computational Effort. While research has established that context-insensitive analyses sacrifice a significant amount of precision, fully context-sensitive analyses demand long computation times, rendering them unpractical. With their incremental algorithm, the authors of *Rethinking Incremental and Parallel Pointer Analysis* [15] could greatly reduce the amortized time spent for updating the Pointer Assignment Graph of a context-sensitive analysis. They save time after code deletions by skipping unaffected parts of the graph. However, in unfavorable cases, their algorithm still needs minutes to do a single update. Therefore, hybrid context sensitivity is currently a field of study [13] and is employed in practice.

Declarative Specification. PTAs can be categorized as being either *imperative* or *declarative*. In the *Doop* framework [6], sophisticated analyses can be specified in a declarative way, using Datalog rules. Given a set of initial facts, called the *Extensional Database*, a Datalog engine computes all derivable facts using the analysis rules. This approach accelerates the development of an analysis significantly. However, the run-time performance may suffer compared to a hand-written imperative implementation.

In our case, by using adjacency lists, our graph model can efficiently discover successor nodes. Datalog engines would employ database-join operations to match nodes with their outgoing edges. Additionally, the expressiveness of Datalog rules quickly reaches its limits: efficient representation of our Saturation rule requires *frequency support goals*—comparison with a counting aggregate. Even though these would not break fact derivation monotonicity, they are not supported by standard Datalog engines [17].

9 Conclusions

Debloating software by only shipping code exercised during testing is risky: since a developer is not able to oversee the huge set of removed methods, the reliability in production becomes uncertain. However, only relying on static analysis leaves much of the bloat to remain. Hence, we suggest leveraging analysis results to identify a concise set of impactful methods for effective software debloating.

In order to predict the effects of a cut, we capture the analysis in a directed hypergraph model. To enable interactive computation, we introduce a call graph construction scheme that provides a reasonable trade-off between existing ones. Furthermore, we present an incremental algorithm that speeds up the amortized computation time for each prediction on batches. We show how it can be used both in an interactive setting to guide a developer towards worthwhile cuts, and to compute the next most effective cut in a fully-automated way.

We find that the cognitive and computational overhead introduced by pragmatically modeling the propagation of types through variables pays off in more concise cut suggestions. Finally, as the automatically generated cuts indicate, manually removing only a few methods of a project has significant potential for debloating.

10 Data Availability

We provide our CutTool, our implementation of the graph search algorithm, and the scripts used for evaluation as an artifact [4]. In order to not require building our fork of GraalVM Native Image, we additionally provide the collected graph files for the example projects.

Acknowledgments

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC '19)*. ACM. https://doi.org/10.1145/3359789.3359823

[2] Giorgio Ausiello, Paolo G. Franciosa, and Daniele Frigioni. 2001. Directed Hypergraphs: Problems, Algorithmic Results, and a Novel Decremental Approach. In *Proceedings of the 7th Italian Conference on Theoretical Computer Science (ICTCS '01)*. Springer. https://doi.org/10.1007/3-540-45446-2_20

[3] David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*. ACM. https://doi.org/10.1145/236337.236371

[4] Christoph Blumschein, Fabio Niephaus, Codrut Stancu, Christian Wimmer, Jens Lincke, and Robert Hirschfeld. 2024. *Finding Cuts in Static Analysis Graphs to Debloat Software - Artifact.* https://doi.org/10.5281/zenodo.12669148

[5] 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 *Proceedings of the International Conference on Software Engineering (ICSE '11)*. ACM. https://doi.org/10.1145/1985793.1985827

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

[7] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-Depth Investigation into Debloating Modern Java Applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. ACM. https://doi.org/10.1145/3368089.3409738

[8] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the*

9th European Conference on Object-Oriented Programming (ECOOP '95). Springer. https://doi.org/10.1007/3-540-49538-X_5

[9] Lisa Nguyen Quang Do and Eric Bodden. 2022. Explaining Static Analysis With Rule Graphs. *IEEE Transactions on Software Engineering* 48, 2 (2022). https://doi.org/10.1109/TSE.2020.2999534

[10] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. 1993. Directed Hypergraphs and Applications. *Discrete Applied Mathematics* 42, 2 (1993). https://doi.org/10.1016/0166-218X(93)90045-P

[11] Ara Hayrapetyan, David Kempe, Martin Pál, and Zoya Svitkina. 2005. Unbalanced Graph Cuts. In *Algorithms – ESA 2005 (ESA 2005)*. Springer. https://doi.org/10.1007/11561071_19

[12] Jeff Johnson. 2020. *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Guidelines* (third ed.). Morgan Kaufmann.

[13] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM. https://doi.org/10.1145/2491956.2462191

[14] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. https://doi.org/10.1109/ICSE.2017.53

[15] Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking Incremental and Parallel Pointer Analysis. *ACM Transactions on Programming Languages and Systems* 41, 1 (2019). https://doi.org/10.1145/3293606

[16] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. 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). https://doi.org/10.1145/2644805

[17] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. 2013. Extending the Power of Datalog Recursion. *The VLDB Journal* 22, 4 (2013). https://doi.org/10.1007/s00778-012-0299-1

[18] Micronaut foundation. 2023. Micronaut. https://micronaut.io

[19] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and Exploiting the K-CFA Paradox: Illuminating Functional vs. Object-Oriented Program Analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM. https://doi.org/10.1145/1806596.1806631

[20] RedHat. 2023. Quarkus. https://quarkus.io

[21] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. https://doi.org/10.1007/978-3-030-22038-9_23

[22] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. ACM. https://doi.org/10.1145/3238147.3238160

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

[24] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded*. Springer. https://doi.org/10.1007/978-3-642-24206-9_14

[25] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A Longitudinal Analysis of Bloated Java Dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. ACM, New York, NY, USA. https://doi.org/10.1145/3468264.3468589

[26] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. 2023. Coverage-Based Debloating for Java Bytecode. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023). https://doi.org/10.1145/3546948

[27] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Optimization Coaching: Optimizers Learn to Communicate with Programmers. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM. https://doi.org/10.1145/2384616.2384629

[28] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical Virtual Method Call Resolution for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM. https://doi.org/10.1145/353171.353189

[29] Frank Tip and Jens Palsberg. 2000. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM. https://doi.org/10.1145/353171.353190

[30] VMWare. 2023. Spring. https://spring.io/

[31] Christian Wimmer, Codruţ Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019). https://doi.org/10.1145/3360610

[32] Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. 2024. Scaling Type-Based Points-to Analysis with Saturation. *Proceedings of the ACM on Programming Languages* 8 (2024). https://doi.org/10.1145/3656417