Frankenstein

fast and lightweight call graph generation for software builds

Keshani, Mehdi; Gousios, Georgios; Proksch, Sebastian

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Frankenstein: fast and lightweight call graph generation for software builds

Mehdi Keshani[1] · Georgios Gousios[1,2] · Sebastian Proksch[1]

Abstract

Call Graphs are a rich data source and form the foundation for advanced static analyses that can, for example, detect security vulnerabilities or dead code. This information is invaluable when it is immediately available, such as in the output of a build system. Call Graph generation is a whole-program analysis: not just the application, but also all its dependencies are processed together. Recent work has shown that even advanced static analyses can use summarization techniques to substantially improve runtime; however, existing analyses focus on soundness, and as such remain very expensive. When executed in the build system, which typically has limited resources, even powerful servers suffer from slow build times, rendering these analyses impractical in today's fast-paced development. In this paper, we aim to strike a balance between improving static analyses while remaining practical for use cases that require quick results in low-resource environments. We propose a summarization-based implementation of a Class-Hierarchy Analysis algorithm for call graph generation of Java programs. Our approach leverages the fact that dependency sets often do not change between builds: we can generate call graphs for these dependencies, cache their generation for subsequent builds, and using a novel *stitching* algorithm, Frankenstein, merge all partial results into a complete call graph for the whole program. Our evaluation results show that this lightweight approach can substantially outperform existing frameworks. In terms of speed improvements, Frankenstein surpasses the baselines by up to 38%, requiring an average of just 388 Megabytes of memory. This makes the proposed approach practical for build systems with limited memory resources. Despite these optimizations, our generated call graphs maintain a near-identical set of edges when compared to the baselines, achieving an F_1 score of up to 0.98. This summarization-based approach for call graph generation paves the way for using extended static analyses in build processes.

Communicated by: Sigrid Eldh, Davide Falessi and Burak Turhan

This article belongs to the Topical Collection: *Special Issue on Software Engineering in Practice*.

✉ Mehdi Keshani
  m.keshani@tudelft.nl

Extended author information available on the last page of the article

🖄 Springer

1 Introduction

Continuous integration and delivery have revolutionized modern software engineering. Many tools and analyzers are applied in the build pipelines to enhance developers' productivity (e.g., improving the handling of pull requests (Vasilescu et al. 2015)). However, this comes at a price: increased build time, which hinders build servers from providing fast feedback. Developers have to decide between faster builds or more helpful analyses. Especially in open-source software, developers often use shared, resource-limited build services like GitHub or TravisCI, which require fast and lightweight tools. For instance, DEPENDABOT (Dependabot 2023) conducts relatively basic analysis at the *versioned package* level to identify vulnerable dependencies, which lags behind more advanced state-of-the-art approaches (Hejderup et al. 2018; Boldi and Gousios 2021). While this is less accurate, it is still useful and fast, so developers widely accept the tradeoff. We envision leveraging the vast information on centralized infrastructures like GitHub for ecosystem-scale analyses to provide better support for developers, further emphasizing the need for scalable approaches.

Traditionally, *program analysis* aims at soundness as the key property and is performed through *whole-program analyses*, which are more precise but also expensive. Recent results show that even advanced analyses can use summarization techniques without sacrificing soundness (Schubert et al. 2021). However, despite substantial performance improvements, they are still expensive, which hinders their widespread adoption in practice, especially when high performance is needed. Several thought leaders in the field advocate for relaxed requirements on soundness and favor trading off soundness to make program analyses more relevant in practice (Livshits et al. 2015). Previous work has shown that it is possible to summarize static analyses and pre-compute certain parts of an analysis (Toman and Grossman 2017; Nielsen et al. 2021; Schubert et al. 2021). Recent work even demonstrates that pruning analysis results, while being highly unsound, can have positive effects for users, as it results in faster execution and higher precision of two static analyses (Utture et al. 2022).

Many advanced program analyses, such as the detection of vulnerable call chains or unused code, rely on a *call graph* (CG), an approximation of all call relations between different callables (i.e., methods) of a system. CGs are generated through powerful data and control flow analyses on the whole program, including all of its dependencies. Unfortunately, generating the CG of an average program can easily take minutes, even when only basic algorithms are used. The resulting CG will take up several gigabytes of memory for big programs (e.g., h2o project (2023)), often even more during the generation. This is neither practical for build systems nor for analyses of large codebases. While the former are limited by their resources, analyses of the latter are usually restricted to static information (Dyer et al. 2015) because advanced analyses are beyond existing approaches due to time and memory requirements. We observe, though, that the largest part of a CG originates from dependencies, both direct and transitive. Research has shown that 81.5% of programs keep outdated dependencies (Kula et al. 2018), and it is evident that between the small and frequent changes of two subsequent builds, the likelihood of dependency change is even lower. We propose to leverage this and eliminate redundancy to speed up CG generation. The same idea applies in software ecosystems where a few popular libraries are included in many programs, so caching and reusing their analysis results pays off (Keshani 2021).

In this paper, we investigate the idea of pre-computing *Partial CGs* (PCGs) that store minimal information about isolated dependencies (e.g., their declared types and methods) and reducing the CG generation to combine these partial results in three distinct steps. 1) Resolve all (direct and transitive) dependencies of the program at hand. 2) Generate a PCG for

each (isolated) dependency and remember the type hierarchies defined in the dependency. Similar to the caching of dependencies in a build job (to prevent re-downloading), pre-computed CGs can be cached across builds. They need to be computed once per dependency and can then be reused whenever this particular dependency is used. 3) Merge the PCGs using our novel *stitching* algorithm.

In contrast to previous work on summarization (Schubert et al. 2021), we are willing to relax soundness guarantees and put a strong focus on investigating the trade-offs necessary for a practical approach. Schubert et al. (2021) improve the speed of CG generation using summarization. They evaluate this formally on several projects. However, it is unclear how effective this approach is in practice when analyzing large projects with many or large dependencies. An example of this trade-off is our decision to pick Class-Hierarchy Analysis (CHA) for the CG generation. CHA is a very basic algorithm, but it only needs to preserve the type hierarchy and call site information. This is the least amount of information with which it is possible to generate a CG. The *Reachability Analysis* (RA) algorithm requires even less information, but the usefulness of the resulting CG drastically drops due to high imprecision. This makes CHA the most applicable approach for our use cases.

Advanced approaches, like MODALYZER, can generate more accurate results and, through summarization, decrease CG generation by 90% (they report a reduction from 4h to 25min) (Schubert et al. 2021). However, this takes too long for our use case. Moreover, it is infeasible to preserve the required context information for complex algorithms on shared infrastructure, as the points-to-graph and CG quickly grow to enormous sizes. Existing studies found that practical static analyses are often not sound (Sui et al. 2018) or that pruning static analysis results through machine learning (ML) can result in increased usability (Tripp et al. 2014). While these are fundamentally unsound solutions, they are promising directions to make static analysis more practical. It is necessary to further investigate the trade-off between soundness, precision, and speed. In this study, we investigate these trade-offs using real-world experiments in great detail.

We demonstrate the validity of our approach in an extensive set of experiments that compare our *stitched* results with the state-of-the-art static analysis frameworks OPAL (Eichberg et al. 2018; Helm et al. 2020) and WALA (T. j. watson libraries for analysis 2023). We use them for two tasks: first, we generate PCGs for isolated *versioned packages*. Second, we run them on whole programs to generate a baseline for comparing speed and correctness. Based on our experiments, we have observed that after the cache is filled, our primary use case of performing subsequent builds experiences a significant improvement in execution speed, ranging from an average of 15% to 38%. We further demonstrate that the CGs generated by our approach for the whole program are comparable to those generated by the baselines. A comparison of the resulting sets of callables in the CGs reveals that our approach achieves a precision of 0.93 for OPAL and 0.99 for WALA. Furthermore, it attains a recall of 0.97 for WALA and 0.95 for OPAL. Moreover, we illustrate that, on average, our approach requires between 388 and 432 Megabytes (Mb) of memory to generate a whole-program CG further highlighting its efficiency in real-world applications.

Our manual investigation shows that any deviations in precision and recall are not inherent to our approach and could be fixed with more engineering effort on the partial analysis. We believe that our approach is promising because the minor reduction in recall allows for improvements that make it practical for inclusion in CI tools.

Overall, this paper presents the following main contributions:

– Adopting an existing library summarization technique for CG generation through a novel *stitching* algorithm

– Revisiting existing evaluation methodologies with a focus on correctness, scalability, and memory consumption designed particularly to assess the practicality of our proposed approach
– Extensive evaluation on a real-world Maven sample

We release our tool, the dataset, and the analysis scripts on the artifact page of this paper (Repository of the paper 2023).

2 Background

In this section, we first introduce the terminology that is utilized throughout this study. Subsequently, we provide an overview of existing studies related to our research.

2.1 Terminology

To ensure clarity and precision, we present definitions of the key terms used in this article. It is important to note that some of these terms may have different and overlapping meanings in various contexts. Therefore, we aim to establish a shared understanding of these concepts for the rest of this article. The definitions are as follows:

PROGRAM A *program* refers to a piece of code written in any programming language, regardless of its size or distribution method.

ARTIFACT An *artifact* is a *program* distributed to users, offering features related to a core idea. *Artifacts* may evolve over time, addressing existing issues or adding new functionalities. In Maven, this is referred to as `groupId:artifactId`.

PACKAGE This term is synonymous with *artifact* and can be used interchangeably. *Package* is also commonly used to denote *namespaces* within *programs*. To avoid confusion in this study, we use the term *Java package* when referring to *namespaces*.

VERSIONED- PACKAGE A *versioned package* is a snapshot of an *artifact* at a specific point in time. *Versioned packages* are released on repositories to be used by others. In Maven, this is referred to as `groupId:artifactId:version`.

PROJECT A project is a domain containing multiple packages (`groupId`). While the term *project* may also be used to refer to *artifacts* or *versioned packages* in other contexts, in this study, we consistently use the term *project* to represent the domain holding a `groupId` and containing *packages*.

DEPENDENCY A *dependency* relationship occurs when one *versioned package*, such as *A*, includes another *versioned package*, like *B*, in order to utilize its functionality. In this scenario, *A* is considered the *dependent* program, while *B* is the *dependency*. In Maven, this relationship is established by adding a *dependency tag* to the *pom.xml* file of *A*.

LIBRARY This term is synonymous with a *dependency*, and the two are used interchangeably.

WHOLE PROGRAM A *whole program* encompasses the entire code required for successful compilation, including any *versioned package* listed as a dependency (direct or transitive).

DEPENDENCY SET A *dependency set* consists of the *versioned packages* required for a *program* to compile successfully. The process of determining this set is referred to as *dependency resolution*. The concept of a *dependency set* shares similarities with the *whole-program*

notion, as both involve the collection of all necessary components for successful compilation.

APPLICATION An *application* refers to a *program* that is the current focus of attention. It serves as the root of a dependency set and the starting point for analyses, such as dependency resolution. The *application* is also included in the *dependency set*.

2.2 Related Work

CG GENERATION ALGORITHMS There are various algorithms for constructing CGs, including Tip and Palsberg ([2000](#)); Shivers ([1988](#)); Sundaresan et al. ([2000](#)). The most basic algorithm is a RA which has multiple variants ranging from only considering method names to also considering the signatures of reachable methods. RA has been used in a variety of use cases in the literature, with pioneers such as Srivastava ([1992](#)) and Goldberg and Robson ([1983](#)).

Although RA is sound and fast, it significantly reduces precision. To address this Dean et al. ([1995](#)) proposed *Call Hierarchy Analysis* (CHA) as a more precise extension. This algorithm matches the called method's signature to subtypes of the receiver type and finds implementations of the dynamically dispatched target. While CHA is sound and scalable, it is less precise than other alternatives. It is, however, a good trade-off and the default CG algorithm for most static analyzers. The next step towards greater precision is *Rapid Type Analysis* (RTA), as introduced by Bacon and Sweeney ([1996](#)). RTA utilizes a CHA CG and prunes its edges. Hence, this algorithm has a higher performance overhead compared to CHA. RTA tracks allocated types and removes unreachable edges from the CHA CG to make the CG more precise. It is worth noting that we do not propose a new algorithm in this paper. Instead, we reuse the existing algorithms described here and combine them with other techniques to make them practical for CI tools.

CG ACCURACY Several studies have investigated the soundness and precision of CGs. By definition, a CG is considered sound if it covers all edges that are possible at runtime and precise if it does not contain edges that cannot be observed at runtime. Sound analysis is crucial for many use cases, especially security analysis, to avoid missing any positive cases. However, a low precision may result in many false positive alarms, which can hurt the usefulness of such an analysis. Increasing precision while maintaining soundness is a challenging and computationally expensive task.

A *dynamic* analysis can capture a perfect CG for a program execution by instrumenting the program and storing all observed invocations. However, this result is only valid for this particular program execution. In contrast, *statically* generating a perfect CG is an undecidable problem (Landi [1992](#); Ramalingam [1994](#); Reps [2000](#)). Furthermore, generating a more precise graph often requires a slower CG construction process. Users can choose to trade off precision for soundness and speed. However, some language features, such as reflection, are difficult to approximate in a static analysis, and generating a sound CG can be challenging in these cases. Livshits et al. ([2015](#)) proposed the term *soundy* to describe an analysis that balances soundness, precision, and scalability. Additionally, Sui et al. ([2020](#)) investigated the soundness of state-of-the-art CG constructions by comparing static and dynamic CGs. They used the term *recall* instead of soundness because, unlike soundness, it is not a binary term. Reif et al. ([2019](#)) proposed a benchmark and compared the soundness of existing CG generators in their paper. Their results show that OPAL (Eichberg et al. [2018](#); Helm et al. [2020](#)) is faster than WALA (T. j. watson libraries for analysis [2023](#)), SOOT (Lam et al.

2011), and DOOP (The doop project 2023), but has lower coverage. They did not study other frameworks such as SLAM (Ball and Rajamani 2001) and Chord (2023). In this paper, we do not compare our approach to these studies. However, we are inspired by the concepts of precision and recall introduced in these studies and adopt them to compare CGs generated by different approaches.

IMPROVING EXISTING ANALYSES   There is a category of studies that instead of proposing new algorithms focuses on improving the existing ones from different aspects by utilizing various techniques.

For example, some studies have investigated CG generation of applications versus libraries (Ali and Lhoták 2012, 2013). Ali (2014) extensively investigated the effects of the absent parts of a program during analysis. Reif et al. (2016) explain how different entry point calculation can affect the results of library CG construction and suggest different configurations for *within-application* and *library* CGs.

Other studies investigate incremental static analysis. Souter and Pollock (2001) created the incremental *Cartesian Product Algorithm* (CPA). Tip and Palsberg (2000) improved the scalability of CG generation for large programs. Alexandru et al. (2019) incrementally analyzed the historical representation of multi-revisioned software artifacts source code. Their proposed approach removes redundant computation for similar parts of different versions of an artifact. The approach only updates the changed parts of new versions in the CG.

Toman and Grossman (2017) discuss the challenges of static analysis and provide possible solutions to deal with them together with some example articles. Their paper suggests solving scalability through the pre-computation of library summary, which is very relevant to the idea of our paper. However, they do not provide any implemented solution and only discuss ideas that might solve concrete analysis problems. Unfortunately, CG generation goes beyond a simple merge of summaries and is not trivial. Arzt and Bodden (2016) utilize a similar idea for the data flow analysis of Android applications.

Nielsen et al. (2021) proposed an approach for combining precomputed modules to compute a full CG. This study is highly relevant to our paper and emphasizes the importance of modular CG construction. However, the focus is on JavaScript, which has vastly different challenges than Java. Multiple studies have investigated the summarization techniques in different domains such as data flow analysis and heap analysis (Reps et al. 1995; Sharir et al. 1978; Whaley and Rinard 1999). Gopan and Reps (2007) propose an approach for generating summary information for a library function by analyzing its binary. In this study, the authors do not focus on any particular use case such as CG generation.

Rountev et al. (2006, 2008) summarize the libraries and use them to do Interprocedural Dataflow Analysis on a main program source code. Dillig et al. present a heap analysis (Dillig et al. 2011) technique based on the summarization of functions. Kulkarni et al. (2016) investigated the library summarization idea for CG generation and points-to analysis, which is also closely related to our work. However, that paper has several shortcomings that we address in our paper, including not comparing to any state-of-the-art CG generator, not studying memory consumption, and only using a sample of ten programs. The most related work to this paper has been presented by Schubert et al. (2021). The authors present MODALYZER, which uses summarization to generate CGs for C and C++ programs. The paper shows that the approach does not have to sacrifice soundness while substantially improving the speed of CG generation. However, the evaluation focuses on correctness and closely inspects results for a limited selection of programs. One case is a large project, for which a runtime of 25 minutes is reported. While this improvement of 90% over the 4-hour baseline is impressive, it is still impractical for our intended use cases. In addition to this, qualitative experiments

show for several smaller programs that the approach does not compromise correctness when compared to a whole-program analysis. In our work, we decided to go a different route. First, we employ a much simpler *Class-Hierarchy Analysis* that does not require tracking extensive context information. Second, we decided to trade off soundness for execution speed to achieve scalability. Finally, we believe it is important to understand the applicability of static analysis to *average* programs, so we compile a larger set of real-world *versioned packages* and their dependencies to evaluate our work.

While soundness is an important property of static analysis, relaxing this requirement to achieve faster program analyses has been advocated by thought leaders in the field (Livshits et al. 2015). Many recent works propose tools that are not sound but useful in practice (Sui et al. 2020), like the rising area of machine learning that uses ML for increased usability of static analysis results (Tripp et al. 2014). While these are fundamentally unsound solutions, they are accepted as promising directions to make static analyses more practical. The most recent example of this idea has been proposed by Utture et al. (2022). The idea to prune CGs to make analyses more scalable is highly unsound, but the authors study the effect of the pruning on two example analyses, type casts, and null pointer detection, and show that the observed effect is small. Not all cases can be detected anymore, but the positive effect on performance and actionability that can be achieved through higher precision outweighs the loss in recall.

We are inspired by the studies that we introduced in this category to improve the existing frameworks. To the best of our knowledge, there are no existing studies that focus on the practicality that can be achieved by the summarization of CG generation in real-world scenarios.

3 Approach

In this paper, we present a novel approach for scalable CG generation of Java programs that enables fast analyses, even in resource-limited environments. We call our approach Frankenstein because it is based on the idea of computing partial results that are stitched together to generate the final result. An overview of the general steps of this approach is shown in Fig. 1. In the first step, all dependencies need to be resolved to generate the complete dependency set. For each of these dependencies, a partial analysis result is requested from a caching component, or created if it does not yet exist in the cache. This partial result consists of 1) the PCG of the isolated dependency, which includes all static Call Sites (CSs) that we can find in the bytecode, and 2) all types, their parent type, and their methods that are being declared in this particular dependency. The subsequent step, the *stitching*, is the core contribution of our approach. We first build a *global type hierarchy* (GTH) and merge all the individual type information of the partial results. We then perform a basic *class hierarchy*
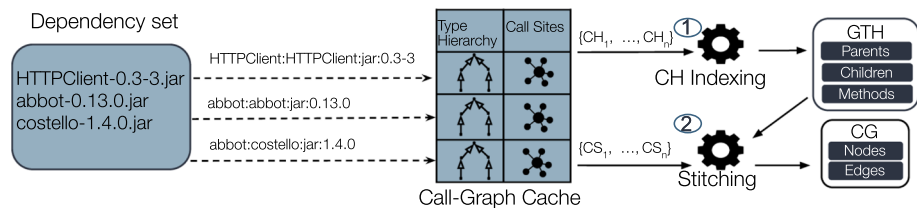

Fig. 1 Overview of CG construction for a project

analysis (CHA) and expand the invocation targets found in the bytecode. This is achieved by adding links to all overriding methods within the matching subtypes that override the original targets.

While the underlying idea is straightforward, CG construction remains a complex task. In the remainder of this chapter, we will elaborate on the individual steps and illustrate the particular challenges and our strategies to overcome them.

3.1 Resolving Dependencies

To generate a CG for an application, it is essential to determine the concrete dependency set for that application through a process known as dependency resolution. In this regard, the widely used build automation tool MAVEN includes a built-in dependency resolver, simplifying the task for developers. Our proposed tool, Frankenstein, relies on the MAVEN dependency resolver to accurately compute the dependency set for an application.

It is worth noting that when defining dependencies in MAVEN, programs can specify version ranges (Maven version ranges 2023). However, this means that dependency resolution may not always be deterministic. A new release of any dependency, either direct or transitive, could potentially alter the resolution result from the perspective of a particular package.

Although the set of available packages may remain stable, the resolution results can vary depending on different contexts. Consider a scenario where a client application *P* has direct dependencies on two *versioned packages*, namely *A* and *B*, both of which transitively rely on a library *L*. However, each of these *versioned packages* requires a different version of *L*, leading to a resolution conflict in Maven. To resolve this, a breadth-first search algorithm is used to select the version of *L* declared "closest" in the dependency tree of *P*. This is because having two versions of the same package in a dependency set simultaneously is not possible. Therefore, the closest version strategy prioritizes the most relied-upon version. Thus, depending on whether *P* depends on *A* or not, the resolved dependency set might differ when viewed from the perspective of *B*.

Overall, there are two challenges that make it unfeasible to simply generate a CG for an application and its dependencies. Firstly, as previously mentioned, the utilization of a specific dependency version is contextual. Secondly, the merging of partial results is not a straightforward task and demands a significant level of interaction between the partial results, which may even change based on the chosen dependency versions.

In Fig. 2, we present an example dependency set and the interactions among its *versioned packages* to illustrate the subsequent steps of our approach. The dependency set comprises three *versioned packages*, and the dependency relations are depicted in the figure. For instance, versions 0 and 1 of dep1 depend on dep2:0. We also demonstrate a time-sensitive dependency resolution in this example, where app:0 specifies a version range dependency on com.example:dep1:[0, 1]. This results in a time-sensitive dependency resolution for app:0. For example, if the resolution is performed at time t1 when the latest released version of dep1 is 0, then dep1:0 would be included in the dependency set. Conversely, if the resolution is done at time t2 after version 1 of dep1 has also been released, then the result would include 1. This time-sensitive dependency resolution highlights the contextuality, the first challenge that we mentioned earlier.

The second challenge occurs at a more granular level, where Dep1 invokes the target() method from Dep2. In a more complex scenario, Dep1 extends Dep2 and overrides the m1() method. In the class App, the object dep is used to call m1(). Depending on the specific type of variable dep, any overridden m1() from sub-classes of Dep2 could be invoked.
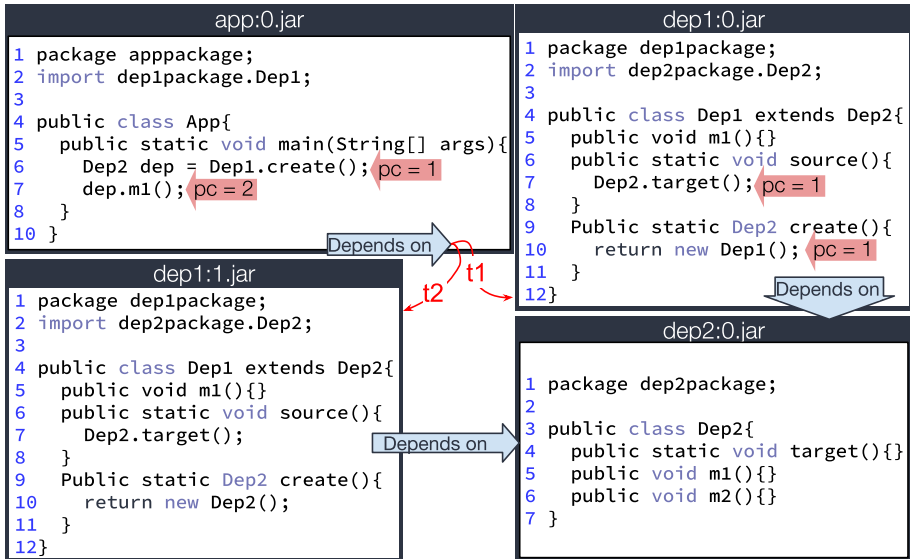
Fig. 2 Example of a dependency set

The exact type of dep depends on the time of dependency resolution. For example, if the dependency resolution occurs in t1, we use dep1:0, and the implemented create() method in this version returns a Dep1 instance, resulting in a call to Dep1.m1() in the App class. However, in t2, when version 1 is used, an instance of Dep2 is returned, and thus the callee of this call would be Dep2.m1(). In this example, the type Dep2 is used to declare the variable dep, which is referred to as the "receiver type". Determining the exact type of the receiver object is a challenging problem that different algorithms attempt to narrow down to varying extents. The CHA algorithm generates a sound CG by including all potential edges from all subtypes, such as Dep2 and its subtypes, including Dep1.

To generate a whole-program CG from partial results the only information that is fixed for a *versioned package* can be stored and it becomes necessary to preserve enough context information about the *versioned package* to allow merging the partial results later in the process. For CG generation this minimal information is the PCG for the isolated dependency and all its declared types and methods. For simplicity, we refer to this data as a PCG. Whenever a whole-program CG should be generated, Frankenstein uses MAVEN for dependency resolution. The resolved dependency set is then used for the subsequent steps in our approach.

3.2 Requesting or Creating PCGs

The next step in generating a whole-program CG is collecting the PCGs for all direct or transitive dependencies of the project in question. We have this dependency set from the previous step and we maintain a basic in-memory key-value store to manage access to the individual results and act as a cache. The key of each PCG is a MAVEN *coordinate* that is composed of a groupId, artifactId, and a version, which uniquely identifies a package within the whole MAVEN ecosystem. If the desired PCG is cached, it can be directly returned, which eliminates redundant processing. However, if the PCG is not yet available in our database, it has to be created first and then added to the cache for future use.

To create the PCGs, we first download the binary (i.e., `.jar` file) of the dependency in question from the MAVEN repository. We then use an existing static analyzer such as OPAL (Eichberg et al. 2018; Helm et al. 2020) to build a CG for this isolated *versioned package* and transform it into a PCG. Different static analyzers may have different configurations and options that allow users to adjust the coverage, accuracy, etc. OPAL for example is highly configurable. We have used it to generate PCGs and, for that, we configure it for library mode, which uses all public methods as entry points in the analysis. OPAL considers that non-private classes, fields, and methods are accessible from outside, non-final classes are extendable and non-final methods are overridable. We have also enabled rewriting of `invokedynamic` instructions to make them easier to analyze. All other configuration options are left at their default values. The complete configuration can be found in our open-source repository to help others reproduce our results.

Using existing frameworks for generating the PCGs has substantial benefits. We do not need to work directly with bytecode and can rely on existing tools. This also enables users to use the framework of their choice as the approach is not dependent on any specific framework. Since existing tools allow for the extraction of the required information for PCG construction, users can use the static analyzer they already have in their CI without adding another dependency to their program.

As touched upon before, PCGs contain two different types of information: a snapshot of the (incomplete) type hierarchy that is declared within the *versioned package*, including the declared methods, as well as the information about CS within the *versioned package*. Figure 3 shows simplified examples of PCGs for *versioned packages* in the example dependency set (resolved in t1).

TYPE HIERARCHY PCGs store the Type Hierarchy (TH) that is defined in a *versioned package*. We use a naming convention similar to Java bytecode to identify types. For example, the type `App` in the *Java package* `apppackage` would be referred to as `/apppackage/App`. We do not store parameters of generic types because due to the type erasure (Bracha et al. 1998) of Java, it is not possible to reason about them. We differentiate between types that
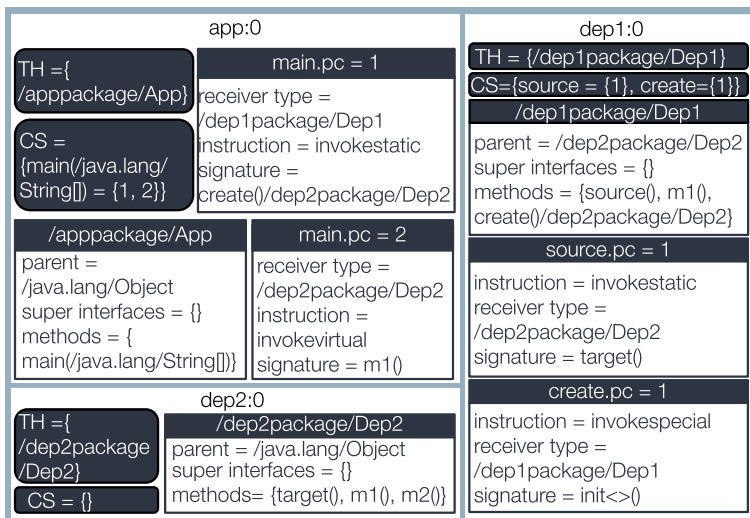

Fig. 3 PCGs of example *versioned packages*

have been declared inside the current *versioned package* (internal) and those in dependencies (external).

For every stored type, we preserve the list of declared methods. As Java supports virtual methods, we can use this later to infer all override relations. We store the method signatures and preserve their name, list of parameters (including types), and the return type. For example, the method signature of the well-known *equals* method `Object.equals(Object)` would be stored as `equals(/java.lang/Object)/java.lang/BooleanType`. For conciseness in Fig. 3 we eliminated the `/java.lang/VoidType` from the methods with no return type.

Our data model allows for the storage of arbitrary meta-data as key-value pairs to cater to the needs of future use cases. We use this, for example, to mark abstract methods without implementation. These arbitrary metadata fields are also removed from the example in Fig. 3 for the sake of brevity.

CALL SITES The minimal information that is required from a CG for later reconstruction is a list of all *CS* that can be found in the PCG. A CS is an instruction in the bytecode that results in a method call. For each CS, we identify the surrounding (*source*) method and the *target* method that is being called, and we store this pair as one call relation. For each call relation, we store the bytecode instruction type, e.g., static invocation.

As Fig. 3 shows, CSs of each source method are indexed by their program counter (*pc*). Since *pc* is unique for each invocation site, we use it as a key. This is helpful for tracing the results; however, it is not necessary for the approach, and one can use any unique key as an index for each CS. For instance, in the case of the *App* class, there are two CS in the main method. In the first one, dep is declared, and in the second one, it is used to call method `m1()`. The receiver type of the first CS is `/dep1package/Dep1` and it is used to call the `create()` method. In the second CS, `/dep2package/Dep2` is the receiver type since it is the type of the variable dep. The first call uses `static` invocation while the second one uses `virtual` invocation. For each CS, we also store the signature of the target method. In the previous example, the first call's signature is `create()/dep2package/Dep2` and the second one is `m1()`. As mentioned before, we use existing tools, more specifically already implemented CHA algorithms to build the PCGs. We only need static information, and existing frameworks and algorithms preserve this information and can be used in this phase. Only receiver type, invocation instruction, and target signature are required, which are available in the bytecode itself without any additional analysis. One could even read the bytecode and build the PCGs directly without using any third-party tool. However, this is beyond the scope of this study. We aim for a lightweight approach that can be easily used on top of existing tools.

Our database fits in regular machines' memory, so we can keep it instantiated for our evaluation. The data model is simple, making it easy to add an efficient, binary disk serialization. Generating the PCGs is expensive, and it is worthwhile to preserve partial results across different analysis executions for any actual use case. For instance, build server integration could preserve partial results from build to build for a substantial speed-up, similar to the caching of MAVEN downloads. Alternatively, a central server can host an in-memory storage of frequently used dependencies across many projects.

3.3 Inferring Global Type Hierarchy

After Frankenstein has acquired all PCGs for the different packages in the resolved dependency set, the next step is to merge the (incomplete) typing information of the individual

Parents index		Defined methods			Children index	
type	**ordered list of parents**	**type**	**signature**	**pkg**	**type**	**set of all children**
/apppackage/App	[/java.lang/Object]	/apppackage/App	main(/java.lang/String[])	app:0	/apppackage/App	{}
			m1()	app:0		
			<init>()	app:0		
/dep2package/Dep2	[/java.lang/Object]	/dep2package/Dep2	target()	dep2:0	/dep2package/Dep2	{/dep1package/Dep1}
			m1()	dep2:0		
			m2()	dep2:0		
			<init>()	dep2:0		
/dep1package/Dep1	[/dep2package/Dep2, /java.lang/Object]	/dep1package/Dep1	m1()	dep1:0	/dep1package/Dep1	{}
			source()	dep1:0		
			create()/dep2package/Dep2	dep1:0		
			<init>()	dep1:0		
			m2()	dep2:0	/java.lang/Object	{/apppackage/App, /dep2package/Dep2, /dep1package/Dep1}
			target()	dep2:0		

Fig. 4 Global Type Hierarchy of example *versioned packages*

versioned packages into a (complete) global type hierarchy. This creates the full picture of the type-system that is used when executing the whole program. We call this a *Global Type Hierarchy (GTH).*

Assuming that the dependency set is complete, merging the individual type hierarchies can be reduced to joining the sets of internal types stored in the individual *THs*. As an example, consider the example dependency set in Fig. 2 (resolved in t1). This set is $DepSet = \{app:0, dep1:0, dep2:0\}$ hence $GTH = TH_{app:0} \cup TH_{dep1:0} \cup TH_{dep2:0}$. All unresolved external types contained in the *THs* will appear as an internal type in one of the other *versioned packages*. For convenience and efficient traversal of type hierarchies, we transform this information into multiple index tables. These index tables are shown in Fig. 4 for the example *versioned packages*. Every type is indexed using its full name since this name should be unique in the classpath of the program otherwise it cannot be compiled. We create three different index tables. One table stores the parents of each type based on the inheritance order. For example, the class /dep1package/Dep1 directly extends /dep2package/Dep2 therefore the first parent that appears in the parent list of /dep1package/Dep1 is /dep2package/Dep2. This sequence continues for each type until we reach the /java.lang/Object class. After the list of all parent classes, we also append a set of all interfaces that a type or any of its parents implement. This is because Java always gives precedence to classes. Also, the order of super interfaces that are appended to the list of parents does not matter because there cannot be two interfaces with default implementations of the same signature in the parent list of a type. The parent index table is not directly used in the *stitching* phase. It is used to facilitate creating the children index and *defined methods index*. Another index that we create is a list of all children of a type. We identify all types that extend or implement a given type, including indirect relationships through inheritance. Indirect relationships occur when a type's ancestor extends/implements the given type, such as a grandparent. This set also does not keep any order. The final index in the *GTH* is the list of methods that each type defines or inherits. The signatures of these methods are then used as another index to find in which *versioned package* they are implemented in. I.e. if a method is not implemented in the current type itself we refer to its first parent that implements it. We use the ordered list of parents in the parent index to retrieve this information efficiently. For example, as shown in Fig. 4 /dep1package/Dep1 inherits method m2() from its dependency dep2:0. Note that since /dep1package/Dep1 overrides m1() we point to dep1:0 as the defining *versioned package* of this signature.
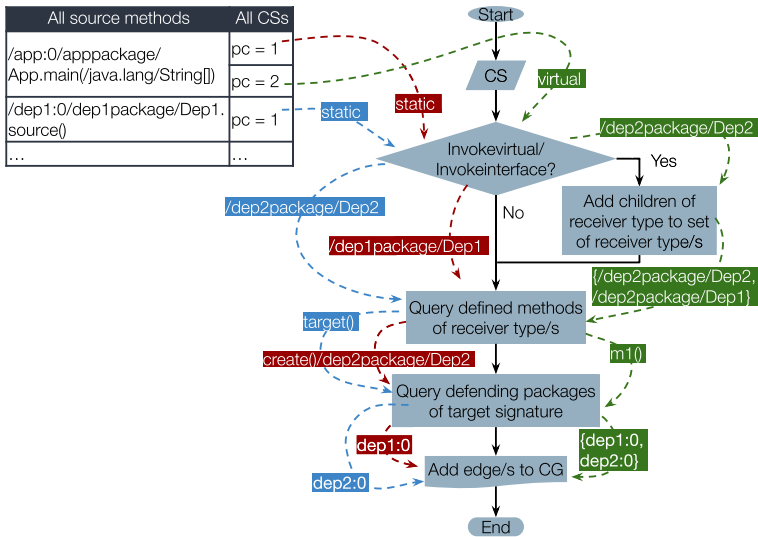
Fig. 5 Resolving edges of CSs

3.4 Stitching the Final CG

Once the PCGs are ready and the global type hierarchy has been established for the complete dependency set, Frankenstein can move to the most crucial part of the approach, the *stitching*. Similar to a compiler that resolves symbols in an abstract syntax tree, we need to connect the CSs that we found in the bytecode with all potential method implementations that could be reached by the corresponding invoke instruction.

The algorithm to achieve this is sketched in Fig. 5. After creating the *GTH* (Section 3.3), the algorithm processes all CSs in all PCGs. The *Java virtual machine* (JVM) supports five different invocation types that require specific handling (`invokestatic`, `invokevirtual`, `invokeinterface`, `invokespecial`, and `invokedynamic`). Depending on the invocation type, Frankenstein selects the correct resolution strategy for the call when processing the CSs. While different invocation types are handled differently within JVM we treat them in two categories of *dynamic dispatch* (`invokevirtual`, `invokeinterface`) and *regular dispatch* (`invokestatic`, `invokespecial`) calls in our algorithm. In the following, we briefly explain these invocation types and how we handle them.

INVOKESTATIC `invokestatic` is used for regular static method calls. This case can be directly resolved by adding an edge from the surrounding method to the static target method from the instruction. As long as the dependency set is a valid resolution, the target type will exist in the GTH and it will always have a matching method declaration signature either implemented in the type itself or one of its parents. A static call `T1.m()` has to have an implementation of `m` in `T1` or one of its parents. If this is not the case, the resolved dependency set is invalid and not executable.

INVOKESPECIAL The `invokespecial` instruction is used for calling non-overridable methods, like private methods, super methods, and constructors. Similar to static invocation, the target type of the CS is in the GTH and can look up the method with a matching signature

to draw an edge. For example, to resolve the constructor call `T2.<init>()`, we lookup `T2` and search for a parameterless constructor.

INVOKEVIRTUAL Instructions that require dynamic dispatch during runtime, `invokevirtual` and `invokeinterface`, are the most challenging to resolve. In Java, the corresponding CSs point to a base type or an interface, but during runtime, the receiving instance could have any possible subtype of this base type. For example, even if the bytecode contains an invocation of the `Object.hashCode()` method, the receiver type might be any type in the GTH that overrides the `hashCode` method. Similarly, while the bytecode might contain a reference to `T3.hashCode()`, `T3` might not override the method, but inherits it from its superclass, in which case the correct target of the invocation is the method declared in the superclass. Frankenstein supports both cases and searches for matching method signatures in all subtypes of the target and draws edges to those that can be called. In case there is none, it finds the first supertype that implements this signature.

INVOKEINTERFACE This invocation is handled in the same way as `invokevirtual` in the JVM. Some optimizations are performed by JVM based on which instruction is used in the bytecode but this does not affect our approach. These optimizations are done on the virtual method table of JVM which we do not use in our approach. Moreover, for `invokeinterface`, it is possible to encounter target methods that are defined outside of the hierarchy of the target type. For illustration, consider a base type B that implements m and an interface I that defines m as well. A subtype S implements I and extends B. As m is inherited, it is not necessary to implement it. For a CG generator, this is a challenging case to handle, as the `invokeinterface` instruction points to I, while the correct target is B.m, despite B having no relation with I. Frankenstein can handle these cases.

INVOKEDYNAMIC Recent versions of the JVM have introduced the `invokedynamic` instruction to support alternate programming languages for the JVM that might use more advanced invocation logic. Resolving such invocations is the most challenging part of CG generation tools. Existing frameworks usually have limited support for these invocations since they require very expensive operation due to their highly dynamic nature. Some frameworks such as OPAL can rewrite these invocations using other invocation instructions. We do not have special handling for these invocations in the *stitching* algorithm. However, if the framework that we use for PCG construction has the feature to rewrite them we utilize it to handle them automatically without special reasoning. As an illustrative example, consider a scenario in which the OPAL framework employs the `invokevirtual` instruction to transform a particular CS that was initially an `invokedynamic` CS. During the bytecode analysis for PCG creation, we store the modified version of the CS. Subsequently, in the *stitching* phase, we utilize the dynamic dispatch handling technique that was previously explained.

Figure 5 shows how we use CS information and GTH to resolve each CS of the example dependency set. Consider the second CS of `App.main` with `pc = 2`. This invocation uses virtual instruction; hence, we need to use the dynamic dispatch category of handling for it. For that, we first query the children index of the GTH to retrieve all children of `dep2package/Dep2`. The result of this query is `/dep1package/Dep1`. Thus we add this child type to the list of receivers. In the next step we query the defined methods within the `/dep1package/Dep1` and `/dep2package/Dep2` using the defined methods index of GTH. Having this information we can easily ask for the exact location of the target method. More specifically we query the result of the previous step for the method signature of `m1()`.

This will identify the two places where `m1()` is defined and are thus potential targets of this call. One target is the implementation within `dep1:0` inside `Dep1` class and the other one is implemented in `dep2:0` in the `Dep2` class. The next CSs will be also processed similarly (see Fig. 5). After all, PCGs have been processed, the resulting CG is ready and can be used for further static analyses.

4 Evaluation

The approach that we discussed in this paper tries to improve the speed of CG generation for consecutive software builds by removing redundant computations. In this section, we investigate how successful the proposed approach is in achieving its goal. We also investigate the effects of the approach on the soundness of CGs and whether it adds any overhead to the memory requirements of existing static analyzers. In a quantitative evaluation, we compare Frankenstein with the current state-of-the-art static analysis frameworks. The evaluation consists of four Research Questions (RQs):

- RQ1: How accurate are Frankenstein's CGs?
- RQ2: How Fast is Frankenstein?
- RQ3: Is Frankenstein generalizable?
- RQ4: How much memory does Frankenstein require?

4.1 Creating a Representative Dataset

Our evaluation strategy for library summarization drew inspiration from previous work by Kulkarni et al. (2016). They evaluated their approach using 10 programs. To improve upon this, we expand the scope of our evaluation by a factor of 5x, using 50 programs in total.

We randomly select 50 packages (i.e., `groupId` and `artifactId`) from the MAVEN repository and then randomly select one version for each package. We use SHRINKWRAP (Shrinkwrap resolvers 2023) to resolve the complete dependency sets for the selected *versioned packages*. SHRINKWRAP is a Java library that re-implements MAVEN'S built-in dependency resolution. During the dependency resolution process, we face "missing artifact on Maven" errors in some cases. In some other cases, dependency resolution results in an empty dependency set, which either means that they were *POM projects* and do not contain code, or they do not include any dependencies. When we face such cases, we discard them and select another random *versioned package* until we have 50 fully resolved dependency sets. These dependency sets include a total of 1044 *versioned packages* (906 unique *versioned packages*). More characteristics of these *versioned packages* are shown in Fig. 6. On average, the selected *versioned packages* have 20 dependencies and consist of 113 files (6051 including dependencies).

In the first two RQs of this paper, we compare Frankenstein and OPAL.[1] To mimic a representative environment, we decided to limit all of our experiments to 7GB of memory, which is the available memory size for `Linux` build jobs on GITHUB.[2] To achieve fairness when comparing two approaches, we remove *versioned packages* for which one of the approaches failed due to OutOfMemoryException. All statistics that we report in the research questions

---

[1] In all of our experiments, we use OPAL version 4.0.0-SNAPSHOT which is included in our artifact page.

[2] https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners#supported-runners-and-hardware-resources, Accessed: 2022-01-15
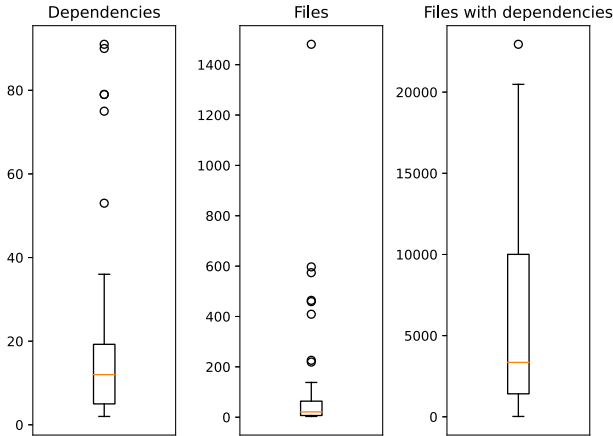
Fig. 6 Selected *versioned packages*' Characteristics

are therefore generated for the dependency sets for which both approaches successfully generated CGs.

4.2 RQ1: How Accurate are Frankenstein's CGs?

The most important assessment that we need to do is to understand whether or not Frankenstein affects the quality of the generated CGs. We are interested in a comparison between the accuracy of CGs generated by Frankenstein and OPAL on its own. Hence, we answer the question of how Frankenstein affects the accuracy of the CGs in the form of two sub-questions:

– How does *stitching* affect the precision of the CGs?
– How does *stitching* affect the recall of the CGs?

METHODOLOGY To investigate the accuracy of the generated CGs by the proposed approach, we compare the generated CGs with OPAL. We take the CGs that are generated by OPAL as the ground truth and do not further investigate their correctness. Previous studies (Reif et al. 2019) have already compared OPAL with other well-known existing frameworks like WALA (T. j. watson libraries for analysis 2023) or SOOT (Lam et al. 2011).

Existing studies (Reif et al. 2019; Sui et al. 2018) measure the soundness of CGs or compare different frameworks. However, they do not aim to compare the edges of real-world programs. Additionally, they do not follow our assumption that OPAL's CGs build the upper bound for our results. We cannot get more accurate than OPAL, since we use it as a base framework for generating the PCGs. Ideally, our approach should be as sound and precise as OPAL. Therefore, we propose an edge-by-edge comparison by calculating the precision and recall of our CGs compared to OPAL's results.

To calculate the precision and recall for each dependency set's CG, we first group all edges of the CG by their source method for both OPAL and Frankenstein. All resulting sets can be identified by type and method name, and we can calculate the precision and recall of Frankenstein CGs compared to OPAL. For the calculation, we construct the intersection between the two results for each source. Assuming that i is the intersection set, f is the set
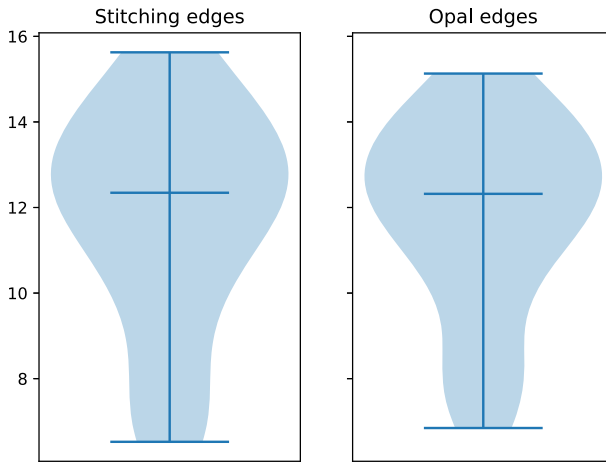
Fig. 7 Edge comparison of Frankenstein and OPAL

of target methods that Frankenstein found for a particular source, and o is the set of target methods identified by OPAL, we calculate precision and recall as follows:

$$precision = \frac{|i|}{|f|} \qquad recall = \frac{|i|}{|o|}$$

For example, let's assume that for the source method `A.src()`, OPAL finds three targets: `B.t1()`, `C.t2()`, and `D.t3()`, while Frankenstein only finds two targets: `B.t1()` and `X.x()`. The intersection set for `A.src()` only contains `B.t1()`; hence, the precision is $\frac{1}{2}$ and the recall is $\frac{1}{3}$.

RESULTS   To investigate the soundness and precision of Frankenstein, we first compare the number of edges generated by OPAL and *stitching*. For 50 dependency sets, OPAL found a total of 25M edges, while Frankenstein found 36M. Figure 7 shows the violin plot of the number of edges. The numbers in this plot are logarithmic. However, these numbers only provide an initial comparison, and it is necessary to further investigate the overlap between the generated edges. We consider two edges identical if the source methods and target methods are identical. We then calculate precision and recall for the targets of each source method as discussed in Section 4.2. A perfect precision of 1.0 means that all stitched targets found for a source are also present in OPAL's CG, while a perfect recall of 1.0 means that all targets in OPAL's CG have also been identified by Frankenstein.

The precision and recall of all source methods in 50 dependency sets are shown in Table 1. In terms of precision, we achieve an average precision of 92.7% using Frankenstein, indicating that most stitched edges can also be found in OPAL's CGs. In terms of recall, there are some cases with a recall of less than 100%, resulting in an average of recall 94.7%. However, the median of precision and recall is still 100%.

Table 1 Precision and recall of Frankenstein

	Mean	Std. Deviation	Median
Precision	92.7%	24.2%	100%
Recall	94.7%	21.4%	100%

These decreases in the recall can be explained by missing CSs in PCGs, which are caused by OPAL not supporting all corner cases in partial analysis or having low coverage. For language features that are not very common, OPAL may not produce complete information in the partial analysis.

To further investigate these problematic corner cases of partial analysis, we manually compared the CGs generated by OPAL and Frankenstein for a benchmark that has been proposed in previous work (Reif et al. 2019). This benchmark consists of a set of annotated test cases that cover numerous possible ways of calling a function in Java. The annotations in these test cases are written by several experts in the field to indicate the expected edges that a sound CG must include. After comparing the two resulting CGs edge by edge, we categorized the CGs of the test cases into four categories:

- $\square_{OF}$: both OPAL and Frankenstein can generate a *sound* CG for the test case.
- $\square_{O}$: only OPAL can generate a *sound* CG for the test case.
- $\square$: neither OPAL nor Frankenstein can generate a *sound* CG for the test case.
- $\boxtimes$: none of the two approaches was able to generate a CG for the test case.

Table 2 shows the results of our manual analysis for the various test cases. The abbreviations refer to different language features, and we refer to the original work for details (Reif et al. 2019). Some language features have multiple test cases to test different ways of calling with that particular Java language feature. In the table, we reported the number of test cases that

Table 2 The results of manual analysis of Stitched CGs

LF	# Cases	$\square_{OF}$	$\square$	$\boxtimes$	$\square_{O}$
CFNE	4	2	0	0	2
CL	1	0	1	0	0
CSR	4	0	1	3	0
DP	1	0	1	0	0
ExtSer	3	3	0	0	0
J8DIM	6	0	0	6	0
J8SIM	1	0	0	1	0
JVMC	5	1	4	0	0
Lambda	4	4	0	0	0
LIB	5	1	1	2	1
LRR	3	0	0	3	0
MR	7	7	0	0	0
NVC	5	1	0	3	1
Ser	9	2	4	3	0
Serlam	2	0	2	0	0
SI	8	3	0	0	5
SPM	7	0	0	7	0
TC	6	0	0	6	0
TMR	8	0	0	8	0
TR	9	0	1	6	2
Unsafe	7	0	0	0	7
VC	4	0	0	4	0
Sum	109	24	15	52	18

we analyzed per language feature in the first column. For example, out of the five test cases that cover the language feature *JVM Calls* (JVMC), four are □, and one is $\square_{OF}$.

In the table, the overall number of test cases in which $\square_{OF}$ approaches generated sound CGs is 24. We also have 52 test cases that ⊠ of the approaches could generate CGs for. This is because we use the library mode of OPAL without providing specific entry points for it due to the nature of our analysis, hence some of these small test cases are not suitable for library analysis because of functions not being reachable from the outside. 15 out of 109 test cases are □ meaning that none of the approaches could generate a sound CG. Since our PCG generation is dependent on OPAL, it is inevitable not to be sound in cases where OPAL itself cannot be sound. The most important cases for our study, however, are the $\square_O$ test cases because they can reveal the reason for the reported 94.7% recall. We further investigate the root cause of these cases to understand whether it is a shortcoming of the approach or not. We have 18 test cases that only OPAL generated sound CGs for. These test cases are distributed among five different language features. The common reason for all missing edges in these test cases is that in the PCG generation phase OPAL does not provide us with all possible CS. This also means that for all CSs that we could extract in the PCG creation phase, we generated the correct edges. Listing 1 shows an example of a reflective call for which OPAL generates a sound CG. This occurs when the whole program is present during the analysis i.e. the Target class is included in the analysis. However, in this example, during partial analysis, the Target class is not present, and OPAL does not provide us with enough information about the *target.target()* CS in the PCG. Since we use OPAL for our partial analysis, this affects the soundness of our approach. Similar to other successful cases, if OPAL had provided the CS information in the PCG, we have could *stitched* it to the Target class. That is, we could not find any cases where the proposed approach inherently causes any unsoundness in these test cases. Nevertheless, the missing CS problem can be fixed with engineering efforts, either on OPAL or by additional handling in the PCG creation phase. We have documented these cases and discussed them with the OPAL developers for future improvements of both tools.

Listing 1 Reflective method call

```
1    public static void main(String[] args) throws Exception {
2        Demo demo = new Demo();
3        demo.field = new CallTarget();
4
5        Field field = Demo.class.getDeclaredField("field");
6        Target target = (Target) field.get(demo);
7        target.target();
8    }
```

4.3 RQ2: How Fast is Frankenstein?

The main benefit of summarization of CG generation is the speedup it provides due to less computation. We need to evaluate whether removing redundant library generation from the process affects the speed of CG generation. It is also important to investigate the extent of any improvements, if any.

METHODOLOGY We design and implement a tool that creates the CG Cache for randomly selected dependency sets. Therefore, we will be able to test the performance of our approach and compare it with the baseline (OPAL).

For this experiment, we need to simulate consecutive builds in software programs. MAVEN hosts a large number of packages with different versions. We select random *versioned pack-*

ages from MAVEN to build their CGs. Hence, we need to resolve the dependency sets of selected *versioned packages*. We use a dependency resolver library called *Shrinkwrap* because it is open source and works well with MAVEN *coordinates*. In some cases, this tool may not be perfect and could result in missing dependencies in the resolved sets. However, we still use it to ease the dependency resolution process, as it is not the main focus of our paper. We ensure that this does not cause any unfairness by always using the same set of dependencies for our approach and baselines.

After selecting the dependency sets, we perform the following steps on each dependency set D:

1. Generate full CGs using Frankenstein:

   - For each *versioned package* in D, we generate a CG using OPAL and parse it into a PCG.
   - We create the CG Cache using the PCGs.
   - For each *versioned package* in D, we fetch PCGs from the CG Cache and then stitch them.

2. We generate a full CG for D using OPAL.
3. Finally, we measure time information and CG statistics of the previous steps to find out how much speed Frankenstein can add in the consecutive generation of the CGs.

RESULTS As shown in Table 3, for these 50 dependency sets on average OPAL takes 6s to generate a CG for each dependency set, whereas Frankenstein takes 5s. The average time for the first round of Frankenstein generation is 18.1s (as shown in Frankenstein$_{initial}$). This round includes PCG generation and caching for the entire set of dependencies, as well as the *stitching* process. However, subsequent rounds benefit from a significant speed improvement. Note that Frankenstein$_{cached}$ represents subsequent rounds of generation after caching and includes the time for generating the PCG of the current version of the application but not the rest of the dependency set. This is because the application's logic usually changes between CI builds, requiring a rebuild of its PCG. All three rows of this table show large standard deviations. This is due to the variety of CG sizes. The fact that multiple dependency sets in the selected sets are considerably larger than others causes longer generation times and hence bigger standard deviations. Generating a PCG for a dependency *versioned package* is a one-time process that needs to be done once and only once. After that, the cached dependencies can be used in different builds. For the first time, Frankenstein may take longer than OPAL due to the combination of caching PCGs and *stitching*. However, in consecutive software builds, Frankenstein saves a lot of time by pre-computing the common parts. Over time, the enhanced speed results in significant time savings, which accumulate with repeated use and translate into considerable overall efficiency gains.

Figure 8 shows the violin plot of the generation time for OPAL, Frankenstein initial generation, and Frankenstein with cached dependencies for 50 dependency sets. The numbers are logarithmic for the sake of better presentation, and median lines are visible for all plots.

Table 3 Time of CG generation different phases (in seconds)

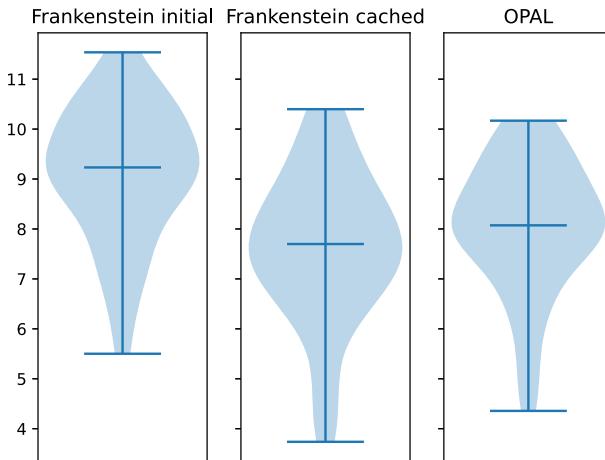|	Mean	Std. Deviation	Median
Frankenstein$_{initial}$	18.1	21.2	10.2
Frankenstein$_{cached}$	5.0	7.4	2.2
OPAL	6.0	6.6	3.2

Fig. 8 Time comparison of Frankenstein and OPAL

Frankenstein with a median of 2.2s outperforms OPAL with 3.2s. It is worth mentioning that Frankenstein's first round of generation with a median of 20.2s is slower than OPAL. This slowdown is because we use OPAL for PCG generation and on top of that we extract the necessary information for PCGs. However, because it is a one-time process the generation time will improve from the second CG generation onwards.

4.4 RQ3: Is Frankenstein Generalizable?

In the previous sections, we presented the benefits of Frankenstein in terms of the speed of CG generation. We also investigated its effects on the accuracy of the CGs. However, since the proposed Frankenstein aims to make CG generation practical for software builds, it is vital to consider the limitations that build servers have to validate the practicality of Frankenstein. On the one hand, Frankenstein requires memory to cache partial results. On the other hand, build servers often have limited memory available. Therefore, we need to investigate how much memory Frankenstein requires.

METHODOLOGY   To investigate whether Frankenstein is generalizable, we slightly adapt our setup and use WALA as the static analysis backend for generating CGs. Apart from this adjustment, we use the same methodology as in RQ1 and RQ2 to compare the speed and accuracy of the generated CGs. Overall, we compare the CGs generated by WALA and Frankenstein for 50 randomly selected *versioned packages*. We use WALA both for the whole-program analysis and the PCG generation, configured to run the CHA algorithm, and using all methods of all classes as entry points. To measure the accuracy of the *stitched* CGs, we use the *precision* and *recall* metrics defined in Section 4.2.

RESULTS   Similar to the evaluation on OPAL, the goal of this experiment is to understand the *speed* and *accuracy* of Frankenstein compared to a plain WALA approach. Table 4 shows the results for precision and recall of the generated CGs. The results confirm the same effects for WALA CGs that we have seen before in Table 1. On average, Frankenstein achieves 99% precision, meaning that virtually all identified edges also exist in the WALA

Table 4 Precision and recall of Frankenstein when generating with WALA

	Mean	Std. Deviation	Median
Precision	98.9%	0.07%	100%
Recall	97.5%	11.0%	100%

CGs. Similar to the OPAL experiment, Frankenstein also misses some edges in comparison to WALA, resulting in a recall of 97.5%. When we investigated these cases manually, we found some corner cases that require special handling during CG generation. For example, WALA contains hard-coded information about Java-related classes (such as Object) and functions (such as Object.hashcode()), resulting in additional edges that Frankenstein does not have. Since we did not provide any additional information to Frankenstein about Java-related classes, it is not surprising that it cannot reproduce these edges. These limitations can be addressed in the future by adding special handling for said Java classes or by providing such classes in the dependency set. A second explanation that we found for missing edges is that existing static analyzers are not perfect and might create incorrect edges. In our manual analysis, we also found a case in which classes were copied from a dependency and were then modified locally. When ignoring the origin of a class file, the name of the class is the same locally and in the dependency. As a result, WALA cannot distinguish the classes and adds additional nodes and edges to the dependency class. Since the including .jar file is part of Frankenstein's fully-qualified names, we can avoid these edges. Across the manually inspected cases, we could not find any differences indicating any conceptual limitations, and all deviations could be addressed through more engineering effort. Most importantly, all limitations come at our own cost, and we believe that the results present a fair comparison. Overall, Frankenstein achieves 98.2% F1 compared to WALA.

Regarding our performance investigation of Frankenstein, we observed similar improvements for WALA (Table 5) and OPAL (Table 3). Both tables show substantial speed improvement once the PCGs are cached. On average, WALA generates a CG in 16.6s, and Frankenstein generates it in 10.2s. This means that caching the PCGs speeds up the process by 38% on average. The caching mechanism applies only to dependencies, not to the application itself. This means that the PCG generation time for the application *versioned package* is included in the Frankenstein $_{cached}$. It is important to realize that the first round of generation is slower due to additional analysis to create the PCGs (119.8 seconds). However, this slowdown pays off in the next rounds of CG generation by saving a lot of time and resources. Compared to the OPAL results, the numbers are generally a bit higher, which can be mostly explained by WALA generating a larger CG that has more nodes and edges. However, the relative comparison and performance increments are comparable.

Overall, our results show that Frankenstein is a generic solution. Users can choose a static analysis framework that fulfills their needs, such as OPAL or WALA. Using Frankenstein then allows for substantial speedup of CG generation through pre-computing and caching PCGs with minimal side effects on accuracy.

Table 5 Time of CG generation different phases (in seconds) with WALA

	Mean	Std. Deviation	Median
Frankenstein $_{initial}$	119.8	147.8	65.0
Frankenstein $_{cached}$	10.2	3.3	9.8
WALA	16.6	11.4	12.7

4.5 RQ4: How Much Memory Does Frankenstein Require?

In the previous sections, we presented the benefits of Frankenstein in terms of the speed of CG generation. We also investigated its effects on the accuracy of the CGs. However, since the proposed Frankenstein aims to make CG generation practical for software builds, it is vital to consider the limitations that build servers have to validate the practicality of Frankenstein.

METHODOLOGY   To answer the third research question we investigated memory usage from two perspectives.

First, we imposed memory constraints on all of our experiments. In the CG generation steps detailed in Section 4.3 (specifically the first and second steps), we restricted the memory available to the JVM. This meant that both our baselines and the CGs generated by Frankenstein for the chosen *versioned packages* (as described in Section 4.1) operated within a limited-resource environment, simulating a build server. Following the generation steps, we compared their success rates and the frequency of OutOfMemoryException. This allowed us to examine the impact of memory limitations in real-world scenarios on our proposed method.

Second, we employed the open JDK profiling tool JOL (Jol 2023) to measure the space occupied by Frankenstein within the JVM heap. Specifically, we determined the sizes of the PCGs and index tables (refer to Section 3.3) utilized in the *stitching* process.

RESULTS   After running the experiment, we realized that out of 50 dependency sets, both OPAL-based Frankenstein and OPAL encountered memory errors for 3 *versioned packages*. The fact that they had the same number of memory exceptions indicates that Frankenstein does not introduce any additional memory requirements or add memory overhead to the base framework OPAL. When generating with WALA, however, WALA-based Frankenstein faced fewer memory exceptions compared to WALA itself. WALA failed in 8 instances, while Frankenstein failed in 4 cases due to OutOfMemoryException. This lower memory demand may be attributed to Frankenstein holding just enough information about each *versioned package* to perform a basic CHA algorithm, whereas WALA, as a general static analysis framework, needs to capture a more comprehensive view of the entire dependency set's bytecode.

To assess the memory requirements of Frankenstein for generating a complete program CG, we examined all data necessary for the *stitching* process. For each dependency set, Frankenstein initially caches all dependency PCGs, followed by the creation of *GTH* each time *stitching* is executed. Table 6 displays the memory usage in megabytes for these Data Structures (DSs) in both OPAL-based and WALA-based implementations of Frankenstein. The *PCGs* row is determined by summing the sizes of all PCGs for each dependency set. We then calculated the mean, median, and standard deviations for the selected 50 dependency sets, as shown in the table. Likewise, the *GTH* row is based on the 50 *stitching* operations performed on the selected dependency sets, with the mean, median, and standard deviations reported. In summary, on average, OPAL-based Frankenstein necessitates a total of 432 Mb

Table 6 Memory required by Frankenstein (in Mb) to *stitch* a whole program

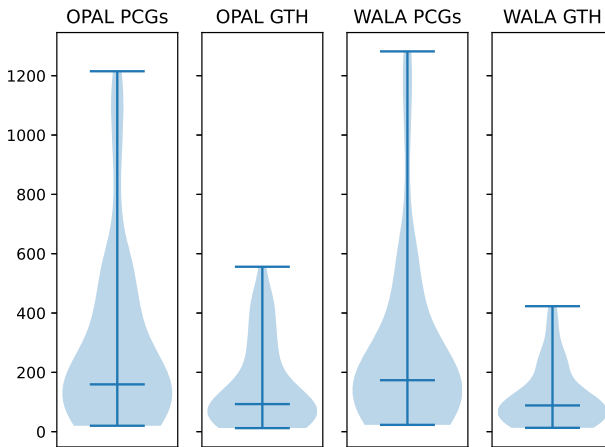|	DS	Mean	Std. Deviation	Median
OPAL-based	**PCGs**	285	299	159
	GTH	152	146	93
WALA-based	**PCGs**	272	301	173
	GTH	116	102	88

Fig. 9 Distribution of Data Structure sizes (in Mb) required by Frankenstein

of memory, while WALA-based Frankenstein demands 388 Mb of memory to generate a whole-program CG. Figure 9 further illustrates the distribution of this data.

An interesting observation from this table is that both WALA-based and OPAL-based figures are within the same order of magnitude. This outcome is somewhat expected since the general approach is similar. However, differences may arise from two factors. One reason is the variation in successfully generated dependency sets. As previously mentioned, some dependency sets did not have a CG generated due to memory exceptions. This implies that the sets of projects successfully generated for WALA-based and OPAL-based Frankenstein are not identical. Another reason for the discrepancies is that the outputs of different frameworks are not identical, even for the same operations. For instance, as we discussed in previous sections the CGs generated by WALA are generally larger than those generated by OPAL due to the different coverages of these tools. This may result from numerous differences in the implementations of the various frameworks.

5 Discussion

As manual inspection has revealed that the lack of recall is mainly due to missing CS information during PCG construction or the internal handling of special cases within WALA or OPAL such as built-in handling of Java-related classes. Nevertheless, we are confident that this limitation can be fixed through engineering work. We strongly believe that the proposed approach has the potential to be adopted by engineers, software companies, and researchers to implement fast and lightweight CG-based analyses that are practical for software development pipelines such as CI tools. Moreover, our proof of concept Frankenstein shows that summarization works in practice. Thus we highly recommend that developers of existing static analysis frameworks adopt the summarization technique that we propose. These tools can benefit from the speed up while reusing their handling of special cases. This potentially can remove or mitigate the accuracy penalties that we reported in this paper.

Our results confirm previous findings in the field. First, as pointed out by Toman and Grossman (2017), library pre-computation can effectively solve the scalability problems of the analysis. Our study validates this idea, particularly for CG generation. However, applying

this technique to different static analysis tasks can have similar positive effects as examined by Arzt and Bodden (2016) for data flow analysis.

We believe that the concept of summarization will strongly impact the design of future static analysis tools. Although configurability and modularity are essential requirements for static analyzers it is important to consider them not only for program design but also for data flow design. Currently, these tools can be highly configurable and different modules can be combined in a pipeline based on specific requirements. For a given configuration, this flexible pipeline of modules is created and executed on top of a single input data, producing a single output. We propose that by modularizing the intermediate data, analysis modules can save the current state at specific points and restart the task from the point of interruption. This approach would make the summarization of different static analysis tasks straightforward. We believe that this will make static analysis more practical for daily usage such as in CI tools.

In this paper, we acknowledge the tradeoff between memory, speed, and cache utilization when using Frankenstein. The caching mechanism significantly improves speed by reusing precomputed PCGs but requires memory for cached PCGs. Our experimental results illustrate these tradeoffs, providing a practical evaluation of Frankenstein's real-world performance improvements and its memory usage. This information will help developers make informed decisions based on the benefits and costs of the tool.

6 Threats to Validity

INTERNAL VALIDITY WALA and specially OPAL offer a plethora of configuration options, all of which may affect our results. To mitigate this we attempted to utilize the maximum coverage that these frameworks provide, such as using the most comprehensive entry point strategies or the assumption of library openness (Reif et al. 2016). Additionally, we provide public access to our code including an aggregated configuration file on our artifact page to ensure a fixed and transparent configuration throughout the study.

Our approach supports all Java invocations except `invokedynamic`. We use OPAL's `invokedynamic` rewrite feature, which rewrites this type of method invocation in the form of other invocations. However, some frameworks may not support the `invokedynamic` rewrite feature, requiring the implementation of heuristics to support `invokedynamic` when adopting our approach. Nonetheless, this and other corner cases that we reported earlier in this paper result in minor accuracy penalties that can be mitigated if developers of existing frameworks adopt the Frankenstein solution within their frameworks. They can reuse the same treatment that they already have in their toolbox for these cases while benefiting from the advantages of Frankenstein.

Since we have implemented multiple tools and conducted various experiments for this study any hidden bug in our implementation can affect our results. To mitigate this threat, we ran our tools on a large data set of real-world *versioned packages*, addressing bugs as we discovered them. Additionally, we implemented an extensive test suite for our tools and programs.

EXTERNAL VALIDITY In this study, we rely on the WALA and OPAL static analysis frameworks. We chose these tools because they are widely used, actively maintained, and we could contact their maintainers. However, this choice may result in inheriting their potential flaws. To mitigate this, we support more than one framework in our approach and designed our intermediate representation (PCGs) in a generic way. Thus, they can be extracted from the

output of other frameworks as well. The information required in PCGs is present in the byte-code and does not require further analysis. Therefore, one could even construct them directly from the bytecode.

7 Future Work

This paper has introduced a summarization-based algorithm for CG generation. In this study, we used WALA and OPAL as the basis of our approach and as a baseline. During this study, we observed that the difference between different frameworks is significant in practice, to the extent that even for the most basic algorithm their outputs differ significantly. Therefore, we believe that it is valuable to investigate these differences and their effects on Frankenstein as future work. For example, we can explore the size of the difference between OPAL and WALA, and how the *stitched* version of these two differs.

The improvements in the scalability of CG generation also affect downstream tasks. In this study, we found evidence of imperfections in existing frameworks themselves. One direction for future work is to evaluate Frankenstein by comparing the effects of CGs generated by Frankenstein and existing frameworks on downstream tasks. For example, we could inves-tigate whether the vulnerable call chains that Frankenstein can find differ from those found by WALA. Using Frankenstein it is also possible to perform more extensive analyses that require CGs. Hence as a next step, we plan to use this approach to generate a large number of CGs and conduct an ecosystem-wide analysis. It is valuable to understand the current state of vulnerability propagation or change impact in MAVEN CENTRAL. Another direction for future work is to study other languages and investigate whether our proposed approach can be applied to them. We plan to explore the differences and specific challenges that other languages may present for summarization.

8 Summary

Generating CGs is essential for many advanced program analyses. The inherent complexity of CG generation and the high memory requirements challenge scalability, making certain static analysis tasks impractical in resource-limited environments such as hosted build servers. In this paper, we propose a summarization-based approach Frankenstein that makes CG con-struction fast and lightweight. Instead of performing a whole-program analysis, we introduce a summarization-based algorithm that precomputes PCGs for all dependencies of an appli-cation and caches the results. For whole program CG generation, an algorithm stitches all PCGs on demand. In our extensive evaluation, we investigate the impact of Frankenstein on accuracy, speed, and memory usage. Our results are very promising when compared to the state-of-the-art approaches. Our results demonstrate that Frankenstein has a minimal impact on the soundness of the generated CGs and can achieve an F1 score of up to 98%. Addition-ally, we show a significant improvement in performance, with up to 38% lower execution time on average and memory usage of only 388 megabytes for the whole program analysis. We believe that the proposed approach in this paper addresses the current key challenges that limit the practicality of static analyzers. Our results show that pre-computing partial static analysis results has a high potential to improve CG generation. We propose a novel algorithm that is the first step towards scalable static analysis in CI tools.

Data Availability We have released our tool, the dataset, and the analysis scripts on the artifact page of this paper (Repository of the paper 2023).

Declarations

Conflicts of interests The authors have no conflict of interest.

References

Alexandru CV, Panichella S, Proksch S, Gall HC (2019) Redundancy-free analysis of multi-revision software artifacts. Empir Softw Eng 24(1):332–380. https://doi.org/10.1007/s10664-018-9630-9

Ali K (2014) The Separate Compilation Assumption. Ph.D. thesis. University of Waterloo, Ontario, Canada. https://hdl.handle.net/10012/8835

Ali K, Lhoták O (2012) Application-Only Call Graph Construction. In: Noble J (ed) In the proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP, Beijing, China. Lecture Notes in Computer Science, vol 7313. Springer, pp 688–712. https://doi.org/10.1007/978-3-642-31057-7_30

Ali K, Lhoták O (2013) Averroes: Whole-Program Analysis without the Whole Program. In: Castagna G (ed) In the proceedings of the 27th European Conference on Object-Oriented Programming, ECOOP, Montpellier, France. Lecture Notes in Computer Science, vol 7920. Springer, pp 378–400. https://doi.org/10.1007/978-3-642-39038-8_16

Arzt S, Bodden E (2016) StubDroid: automatic inference of precise data-flow summaries for the android framework. In: Dillon LK, Visser W, Williams LA (eds) In the proceedings of the 38th International Conference on Software Engineering, ICSE, Austin, TX, USA. ACM, pp 725–735. https://doi.org/10.1145/2884781.2884816

Bacon DF, Sweeney PF (1996) Fast Static Analysis of C++ Virtual Function Calls. In: Anderson L, Coplien J (eds) In the proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA, San Jose, California, USA. ACM, pp 324–341. https://doi.org/10.1145/236337.236371

Ball T, Rajamani SK (2001) Bebop: a path-sensitive interprocedural dataflow engine. In: Field J, Snelting G (eds) In the proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE, Snowbird, Utah, USA. ACM, pp 97–103. https://doi.org/10.1145/379605.379690

Boldi P, Gousios G (2021) Fine-Grained Network Analysis for Modern Software Ecosystems. ACM Trans Internet Technol 21(1):1:1-1:14. https://doi.org/10.1145/3418209

Bracha G, Odersky M, Stoutamire D, Wadler P (1998) Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In: Freeman-Benson BN, Chambers C (eds) In the proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA, Vancouver, British Columbia, Canada. ACM, pp 183–200. https://doi.org/10.1145/286936.286957

Chord (2023) A program analysis platform for java. https://www.seas.upenn.edu/~mhnaik/chord/user_guide/index.html. Accessed 15 Jan 2022

Dean J, Grove D, Chambers C (1995) Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In: Olthoff WG (ed) In the poroceedings of the 9th European Conference on Object-Oriented Programming, ECOOP, Århus, Denmark, Lecture Notes in Computer Science, vol 952. Springer, pp 77–101. https://doi.org/10.1007/3-540-49538-X_5

Dependabot. (2023) https://github.com/dependabot. Accessed 15 Jan 2022

Dillig I, Dillig T, Aiken A, Sagiv M (2011) Precise and compact modular procedure summaries for heap manipulating programs. In: Hall MW, Padua DA (eds) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA. ACM, pp 567–577. https://doi.org/10.1145/1993498.1993565

Dyer R, Nguyen HA, Rajan H, Nguyen TN (2015) Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. ACM Trans Softw Eng Methodol 25(1):7:1-7:34. https://doi.org/10.1145/2803171

Eichberg M, Kübler F, Helm D, Reif M, Salvaneschi G, Mezini M (2018) Lattice based modularization of static analyses. In: Dolby J, Halfond WGJ, Mishra A (eds) In the companion proceedings for the ISSTA/ECOOP Workshops, Amsterdam, Netherlands. ACM, pp 113–118. https://doi.org/10.1145/3236454.3236509

Goldberg A, Robson D (1983) Smalltalk-80: The Language and Its Implementation. Addison-Wesley

Gopan D, Reps TW (2007) Low-Level Library Analysis and Summarization. In: Damm W, Hermanns H (eds) In the proceedings of the 19th International Conference on Computer Aided Verification, CAV, Germany, Lecture Notes in Computer Science, vol 4590. Springer, pp 68–81. https://doi.org/10.1007/978-3-540-73368-3_10

h2o project (2023) https://mvnrepository.com/artifact/ai.h2o/sparkling-water-package_2.11/3.26.8-2.4. Accessed 21 Oct 2022

Hejderup J, van Deursen A, Gousios G (2018) Software ecosystem call graph for dependency management. In: Zisman A, Apel S (eds) In the proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER), Gothenburg, Sweden. ACM, pp 101–104. https://doi.org/10.1145/3183399.3183417

Helm D, Kübler F, Reif M, Eichberg M, Mezini M (2020) Modular collaborative program analysis in OPAL. In: Devanbu P, Cohen MB, Zimmermann T (eds) In the proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE, Virtual Event, USA. ACM, pp 184–196. https://doi.org/10.1145/3368089.3409765

T. j. watson libraries for analysis. (2023) http://wala.sf.net/. Accessed 15 Jan 2022

Jol (2023) https://openjdk.org/projects/code-tools/jol/. Accessed 06 May 2023

Keshani M (2021) Scalable Call Graph Constructor for Maven. In: In the companion proceedings of the 43rd IEEE/ACM International Conference on Software Engineering, ICSE Companion, Madrid, Spain. IEEE, pp 99–101. https://doi.org/10.1109/ICSE-Companion52605.2021.00046

Kula RG, Germán DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? - An empirical study on the impact of security advisories on library migration. Empir Softw Eng 23(1):384–417. https://doi.org/10.1007/s10664-017-9521-5

Kulkarni S, Mangal R, Zhang X, Naik M (2016) Accelerating program analyses by cross-program training. In: Visser E, Smaragdakis Y (eds) In the proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, part of SPLASH, Amsterdam, The Netherland. ACM, pp 359–377. https://doi.org/10.1145/2983990.2984023

Lam P, Bodden E, Lhoták O, Hendren L (2011) The Soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop CETUS, vol 15

Landi W (1992) Undecidability of Static Analysis. LOPLAS 1(4):323–337. https://doi.org/10.1145/161494.161501

Livshits B, Sridharan M, Smaragdakis Y, Lhoták O, Amaral JN, Chang BE, Guyer SZ, Khedker UP, Møller A, Vardoulakis D (2015) In defense of soundness: a manifesto. Commun ACM 58(2):44–46. https://doi.org/10.1145/2644805

Maven version ranges. (2023) https://maven.apache.org/enforcer/enforcer-rules/versionRanges.html. Accessed 21 Oct 2022

Nielsen BB, Torp MT, Møller A (2021) Modular call graph construction for security scanning of Node.js applications. In: Cadar C, Zhang X (eds) In the Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, Virtual Event, Denmark. ACM, pp 29–41. https://doi.org/10.1145/3460319.3464836

Ramalingam G (1994) The Undecidability of Aliasing. ACM Trans Program Lang Syst 16(5):1467–1471. https://doi.org/10.1145/186025.186041

Reif M, Eichberg M, Hermann B, Lerch J, Mezini M (2016) Call graph construction for Java libraries. In: Zimmermann T, Cleland-Huang J, Su Z (eds) In the proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE, Seattle, WA, USA. ACM, pp 474–486. https://doi.org/10.1145/2950290.2950312

Reif M, Kübler F, Eichberg M, Helm D, Mezini M (2019) Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In: Zhang D, Møller A (eds) In the proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, Beijing, China. ACM, pp 251–261. https://doi.org/10.1145/3293882.3330555

Repository of the paper. (2023) https://github.com/ashkboos/LightWeightCGs/tree/main

Reps TW, Horwitz S, Sagiv S (1995) Precise Interprocedural Dataflow Analysis via Graph Reachability. In: Cytron RK, Lee P (eds) Conference Record of POPL: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA. ACM Press, pp 49–61. https://doi.org/10.1145/199448.199462

Reps T (2000) Undecidability of context-sensitive data-dependence analysis. ACM Trans Program Lang Syst, TOPLAS 22(1):162–186

Rountev A, Kagan S, Marlowe TJ (2006) Interprocedural Dataflow Analysis in the Presence of Large Libraries. In: Mycroft A, Zeller A (eds) In the proceedings of the 15th International Conference on Compiler Construction, CC, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, Vienna, Austria, Lecture Notes in Computer Science, vol 3923. Springer, pp 2–16. https://doi.org/10.1007/11688839_2

Rountev A, Sharp M, Xu G (2008) IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In: Hendren LJ (ed) In the proceedings of the 17th International Conference on Compiler Construction, CC, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, Budapest, Hungary, Lecture Notes in Computer Science, vol 4959. Springer, pp 53–68. https://doi.org/10.1007/978-3-540-78791-4_4

Schubert PD, Hermann B, Bodden E (2021) Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis. In: Møller A, Sridharan M (eds) In the proceedings of the 35th European Conference on Object-Oriented Programming, ECOOP, Aarhus, Denmark (Virtual Conference), LIPIcs, vol 194. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp 2:1–2:31. https://doi.org/10.4230/LIPIcs.ECOOP.2021.2

Sharir M, Pnueli A et al (1978) Two approaches to interprocedural data flow analysis. In: New York University. Courant Institute of Mathematical Sciences

Shivers O (1988) Control-Flow Analysis in Scheme. In: Wexelblat RL (ed) In the proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Atlanta, Georgia, USA. ACM, pp 164–174. https://doi.org/10.1145/53990.54007

Shrinkwrap resolvers. (2023) https://github.com/shrinkwrap/resolver. Accessed 15 Jan 2022

Souter AL, Pollock LL (2001) Incremental Call Graph Reanalysis for Object-Oriented Software Maintenance. In: In the proceedings of the International Conference on Software Maintenance, ICSM, Florence, Italy. IEEE Computer Society, pp 682–691. https://doi.org/10.1109/ICSM.2001.972787

Srivastava A (1992) Unreachable Procedures in Object-Oriented Programming. LOPLAS 1(4):355–364. https://doi.org/10.1145/161494.161517

Sui L, Dietrich J, Emery M, Rasheed S, Tahir A (2018) On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation. In: Ryu S (ed) In the proceedings of the 16th Asian Symposium on Programming Languages and Systems, APLAS, Wellington, New Zealand, Lecture Notes in Computer Science, vol 11275. Springer, pp 69–88. https://doi.org/10.1007/978-3-030-02768-1_4

Sui L, Dietrich J, Tahir A, Fourtounis G (2020) On the recall of static call graph construction in practice. In: Rothermel G, Bae D (eds) In the proceedings of the 42nd International Conference on Software Engineering, ICSE, Seoul, South Korea. ACM, pp 1049–1060. https://doi.org/10.1145/3377811.3380441

Sundaresan V, Hendren LJ, Razafimahefa C, Vallée-Rai R, Lam P, Gagnon E, Godin C (2000) Practical virtual method call resolution for Java. In: Rosson MB, Lea D (eds) In the proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA, Minneapolis, Minnesota, USA. ACM, pp 264–280. https://doi.org/10.1145/353171.353189

The doop project. (2023) http://doop.program-analysis.org/. Accessed 15 Jan 2022

Tip F, Palsberg J (2000) Scalable propagation-based call graph construction algorithms. In: Rosson MB, Lea D (eds) In the proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications OOPSLA, Minneapolis, Minnesota, USA. ACM, pp 281–293. https://doi.org/10.1145/353171.353190

Toman J, Grossman D (2017) Taming the Static Analysis Beast. In: Lerner BS, Bodík R, Krishnamurthi S (eds) 2nd Summit on Advances in Programming Languages, SNAPL, Asilomar, CA, USA, LIPIcs, vol 71. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp 18:1–18:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.18

Tripp O, Guarnieri S, Pistoia M, Aravkin AY (2014) ALETHEIA: Improving the Usability of Static Security Analysis. In: Ahn G, Yung M, Li N (eds) In the proceedings of the ACM SIGSAC Conference on

Computer and Communications Security, Scottsdale, AZ, USA. ACM, pp 762–774. https://doi.org/10.1145/2660267.2660339

Utture A, Liu S, Kalhauge CG, Palsberg J (2022) Striking a Balance: Pruning False-Positives from Static Call Graphs. In: In the proceedings of the 44th IEEE/ACM International Conference on Software Engineering, ICSE, Pittsburgh, PA, USA. ACM, pp 2043–2055. https://doi.org/10.1145/3510003.3510166

Vasilescu B, Yu Y, Wang H, Devanbu PT, Filkov V (2015) Quality and productivity outcomes relating to continuous integration in GitHub. In: Nitto ED, Harman M, Heymans P (eds) In the proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, Bergamo, Italy. ACM , pp 805–816. https://doi.org/10.1145/2786805.2786850

Whaley J, Rinard MC (1999) Compositional Pointer and Escape Analysis for Java Programs. In: Hailpern B, Northrop LM, Berman AM (eds) In the proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems. Languages & Applications, OOPSLA, Denver, Colorado, USA. ACM, pp 187–206. https://doi.org/10.1145/320384.320400

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Mehdi Keshani is currently a Ph.D. candidate in the Software Engineering Research Group, at Delft University of Technology, Netherlands. His research interests include program analysis and software ecosystems. His current project revolves around dependency management systems and how to make them more intelligent. He was also a key member of a European project called FASTEN. FASTEN aims at bringing method-level analyses such as call-graph-based impact analyses to the hands of ecosystem developers.

Georgios Gousios is the head of research at Endor Labs, where he is helping to make software supply chains more secure. He is also part-time associate professor of software engineering at the Software Engineering Research Group, Delft University of Technology. Before that, he was a researcher at Facebook. He researches in the broader fields of Software Analytics, Machine Learning for Software Engineering and Dependency Management.

Sebastian Proksch is a researcher interested in demystifying the software development process, both from the perspective of individuals and of teams. He studies the impact of novel technologies and envisions new tools to support software engineers in their day to day tasks. His most recent works in the CI/CD area have created tools that facilitate the adoption of CI/CD and help developers to spot anti-patterns in their build pipelines. Sebastian Proksch is an Assistant Professor in the Software Engineering Research Group at Delft University of Technology (Netherlands) and was the scientific coordinator of the FASTEN project.

Authors and Affiliations

Mehdi Keshani[1] · Georgios Gousios[1,2] · Sebastian Proksch[1]

Georgios Gousios
g.gousios@tudelft.nl

Sebastian Proksch
s.proksch@tudelft.nl

[1] Delft University of Technology, Delft, Netherlands

[2] Endor Labs, Palo Alto, CA, USA