DOI: 10.15514/ISPRAS-2026-38(3)-6

Call Graph Construction for Program Analysis

[1,2] *Fartygin A, ORCID: 0009-0002-2966-6503 <artyom.fartygin@ispras.ru>*
[1] *Borodin A.E., ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1] *Dvortsova V.V., ORCID: 0000-0002-7424-8442 <vvdvortsova@ispras.ru>*
[1] *Afanasyev V.O., ORCID: 0000-0002-8036-0633 <vafanasiev@ispras.ru>*
[1] *Galustov A.L., ORCID: 0009-0001-9591-5873 <artemiy.galustov@ispras.ru>*
[1,3] *Belevantsev A.A., ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *Moscow Institute of Physics and Technology, 9 Institutskiy Pereulok, Dolgoprudny, Moscow Oblast, 141701, Russia.*

[3] *Lomonosov Moscow State University, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. This paper considers a task of call graph construction for programs written in different programming languages including C, C++, Java, Kotlin, Scala, Go, Python. We evaluate the characteristics of various open-source call graph construction tools and then present the design and implementation of our own tool, which supports all these languages. Our approach details key implementation strategies, such as build interception, intermodular linkage, and devirtualization. Finally, we provide experimental results that compare the performance of our tool with that of similar open-source solutions.

Keywords: Static analysis; call graph; linkage graph; JVM; C/C++; Python; Golang; CodeSkeleton; Svace.

For citation: Artem Fartygin, Alexey Borodin, Varvara Dvortsova, Vitaly Afanasyev, Artemiy Galustov, Andrey Belevantsev. Call Graph Construction for Program Analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 3, part 1, 2026, pp. 115–128. DOI: 10.15514/ISPRAS–2026–38(3)–6.

Acknowledgements. The results were obtained using the services of the Ivannikov Institute for System Programming (ISP RAS) Data Center.

Построение графа вызовов для анализа программ

[1,2] *Фартыгин А., ORCID: 0009-0002-2966-6503 <artyom.fartygin@ispras.ru>*
[1] *Бородин А.Е., ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1] *Дворцова В.В., ORCID: 0000-0002-7424-8442 <vvdvortsova@ispras.ru>*
[1] *Афанасьев В.О., ORCID: 0000-0002-8036-0633 <vafanasiev@ispras.ru>*
[1] *Галустов А.Л., ORCID: 0009-0001-9591-5873 <artemiy.galustov@ispras.ru>*
[1,3] *Белеванцев А.А., ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Московский Физико-Технический Институт, 141701, Россия, г. Долгопрудный, ул. Первомайская, д. 5.*

[3] *Факультет вычислительной математики и кибернетики МГУ им. М.В. Ломоносова, 119991, Россия, г. Москва, ул. Колмогорова, д. 1, стр. 52.*

Аннотация. Статья посвящена задаче построения графа вызовов для программ, написанных на популярных языках программирования (C/C++, Java, Kotlin, Scala, Go, Python). Описываются открытые инструменты, поддерживающие создание графа вызовов программы на одном из этих языков, и предлагается собственный инструмент определения графа вызовов, который поддерживает все перечисленные языки. Описываются ключевые компоненты предложенного инструмента, в частности, перехват сборки и межмодульная компоновка для точного построения внутреннего представления программы, а также применяемые алгоритмы девиртуализации. Представляются экспериментальные результаты работы инструмента на ряде проектов с открытым исходным кодом, а также результаты работы открытых инструментов на тех же проектах.

Ключевые слова: статический анализ; граф вызовов; граф компоновки; виртуальная машина JVM, язык программирования C/C++, язык программирования Python, язык программирования Go, программный инструмент CodeSkeleton, анализатор Svace.

Для цитирования: Фартыгин А., Бородин А.Е., Дворцова В.В., Афанасьев В.О., Галустов А.Л., Белеванцев А.А. Построение графа вызовов для анализа программ. Труды ИСП РАН, том 38, вып. 3, часть 1, 2026 г., стр. 115–128 (на английском языке). DOI: 10.15514/ISPRAS–2026–38(3)–6.

Благодарности. Результаты получены с использованием услуг Центра коллективного пользования Института системного программирования им. В.П. Иванникова РАН – ЦКП ИСП РАН.

1. Introduction

To correctly implement interprocedural analysis, the analyzer needs to have information about which function is called when its name is encountered in the source code, which is conveniently represented in the form of a *call graph*. A call graph describes knowledge about call points and which procedures were called. Each node in such a graph represents a program procedure. The graph itself is directed, and an edge from node *A* to node *B* indicates that procedure *A* calls procedure *B*.

We now examine the applications of call graphs across different static analysis methods. Some static analyzers [1, 2] implement summary-based static analysis. A *summary* is a short description of a function's behavior used at the function's call points. With this type of analysis procedures are selected by traversing call graph from bottom up, that is, from callees to callers. Notably, this type of analysis has high parallelization potential since procedures that are not connected in call graph can be analyzed independently. It also scales well to whole program analysis since each procedure only needs to be analyzed once.

Another example of call graph usage is the set of helpful features provided by modern IDEs. For instance, in Visual Studio Code [3] and IntelliJ IDEA [4] a call graph is used internally to power features such as "Find Usages" / "Find All References", "Go to Definition", "Go to Implementation", "Call Hierarchy", "Dead Code Detection", and "Refactoring".

Although call graph itself is rarely displayed visually as a diagram (except in specialized tools), its underlying logic and data form the foundation of these key IDE capabilities.

A call graph can also be used to analyze a software project's dependencies. The goal of this type of analysis is to find third-party dependencies that may be dangerous to use. This analysis can be performed by reading manifests with project's dependencies, but the presence of a dangerous dependency in a manifest does not always mean it is used in a project. It is important to understand which library procedures may be called. This is where a call graph can help verify the reachability of such calls. Reachability of a call to a vulnerable procedure in the call graph can serve as evidence that the library is being used and can help pinpoint specific vulnerabilities in the software project.

This paper describes our experience for constructing call graphs. We implemented a call graph construction tool called CodeSkeleton, which is a component of the Svace static analyzer [5, 6] performing the task of constructing a call graph for further analysis.

2. Existing solutions

Currently, a number of tools and frameworks are available for constructing call graphs in different programming languages. This section provides an overview of these existing solutions.

Most of the solutions we reviewed implement classic devirtualization algorithms for call graph construction: Class Hierarchy Analysis (CHA) [7, 8], Rapid Type Analysis (RTA) [7, 8], Variable Type Analysis (VTA) [8], and Pointer Analysis (PA) [9]. These tools also share the lack of automatic interception of the project build being analyzed. This is critical for analyzing large projects consisting of many components [10-12]. Such severe limitations require rewriting the build scripts to make the tool usable, which is a non-trivial task.

For C/C++, the LLVM framework [13] is worth mentioning. One can try writing one's own pass to traverse the Abstract Syntax Tree (AST) or intermediate representation (IR) being built, but it's not a ready-made solution suitable for subsequent analysis. It's also possible to build a call graph using the opt [14] tool, but the output will be in an image format, making it unsuitable for call graph analysis without additional details.

SootUp [15] is a static analysis framework for JVM bytecode. Among other functionality, it allows to build call graph from provided bytecode. Different call graph building algorithms can be used, including CHA, RTA and PA. There is also a WALA [16] framework, which allows to analyze JVM bytecode and build call graphs in SootUp fashion [17]. Such frameworks may be used for performing various static analysis algorithms in various languages or for inspecting bytecode for other purposes, but they can't be used standalone since they can't automatically detect compiled bytecode (it must be provided manually or by some other tool that intercepts compilation commands).

When considering tools for programs written in Golang (Go), existing solutions are mostly based on the SSA (Static Single Assignment [18]) form from the go-tools [19] package as their IR. The go-tools package already implements call graph analysis algorithms, including CHA, RTA, VTA [20], and PA [21] (now deprecated [22]).

We have reviewed the following analyzers, identifying their strengths and weaknesses:

- callgraph [23] (It uses CHA, RTA, and VTA analyses).
- go-callvis [24] (It uses only a deprecated pointer analysis [21]).
- gocyto [25] (It uses CHA, RTA, and VTA analyses, but was last updated five years ago).

A major drawback of all these tools is that they cannot be run without modifying the build script, which is especially problematic for large projects such as Kubernetes [11].

When searching for Python (more specifically python3) callgraph tools we encountered three tools, namely PyCG [26], pythoncallgraph [27] and Pyan3 [28]. Out of these the most interesting is PyCG, which constructs an assignment graph to correctly evaluate calls that can occur when different OOP or functional programming patterns are used. Other tools perform heuristic-based analysis, mostly

matching functions and methods by name. Unfortunately, we were unable to evaluate any of these tools because (at the time of writing) PyCG is severely outdated and unmaintained and the other two despite some recent work fail to work on even the simplest projects including their own source code as they don't yet support all of the language features available in new python versions.

There are several tools like CodeQL [29] and Joern [30] that allow to collect different code properties and analyze them (e.g., gathering source code metrics or performing taint analysis). Static call graph is one of the code properties which can be built by these tools (using CHA in Joern and PA in CodeQL).

3. The Svace static analyzer

The Svace static analyzer performs deep interprocedural analysis of programs to find errors in the source code. Svace supports more than 10 languages with the main analysis engine working with programs written in C, C++, Java, Kotlin, Scala, Go, and Python; below we will consider the analysis workflow for this engine. The analysis process is divided into two stages:

- Building an IR of a program for subsequent analysis using build interception [31].
- Analyzing the IR and issuing warnings about the errors found.

The goal of the first stage is to create a program representation that accurately reflects how the executable version of the program is built from project's source files. For compiled languages, build interception is the most convenient way to obtain a program representation that accurately reflects user's decisions regarding linking. The analyzer builds the program exactly as the user does, but under its own control, recording all events of interest, primarily, the execution of compilers, assemblers, linkers, and other development tools (a *toolchain*). The resulting records allow the build events to be recreated using the analyzer's own compilers.

In an original build, when a function is accessed by name, the linker resolves external dependencies. For more accurate program analysis, a build interception would also need to track the execution of the linker (or archiver). For example, for C/C++ tracking linkage events allows determining how program files are linked together and resolving function calls from different modules. For JVM languages, similar capabilities are provided by accounting for JAR libraries in compiler calls.

The construction of the local call graph varies for each language. For example, in JVM languages and Python all calls are virtual, while in C/C++ and Go direct, indirect, and virtual calls are allowed. All these specifics will be discussed later; for now, it's important to note that a local call graph is constructed for each module. This will be further refined.

In the second stage of the workflow the collected program representation is analyzed to find errors and vulnerabilities. This analysis can be divided into the following phases:

- Reading local call graphs for each module;
- Creating a global call graph taking into account the program layout;
- Devirtualization to resolve indirect calls;
- Summary-based analysis using the call graph obtained in the previous steps.

4. Linkage

A key auxiliary task in call graph construction is the accurate resolution of program linking. When observing only compilations of individual source files, an analyzer cannot accurately reconstruct a call graph, as it often remains unclear which specific external function is being invoked. In large software systems, this frequently leads to ambiguous situations where multiple candidate functions could potentially be called. Therefore, it is necessary to record information about which program modules were linked together and use this information when constructing the call graph [32]. This

approach ensures an unambiguously accurate representation of the linkage, which constitutes its main advantage.

Let us introduce a concept of a *linkage graph* (LG). Its vertices represent modules, libraries, or executable files, and a directed edge from vertex *A* to vertex *B* indicates that *A* is used in construction of *B*. In this graph, modules are always leaves, while executable files can only appear as parent (non-leaf) vertices. While a call graph captures the program's structure by showing how individual functions may invoke one another, LG establishes a higher-level organization by grouping together modules that can call each other's functions.

The implementation for C/C++ is the most complete, as it incorporates linking information. We now describe the construction of a LG using C/C++ as an example. To construct an accurate LG, it is necessary during a build interception to also capture and process build events that are critical for linking: compilation, assembly, and the creation of both static and dynamic libraries.

Let us take a closer look at the processes that influence the linkage:

1. **Compilation**. The processes of compilation and assembly allow us to obtain information about the name of the source code file, the object code file, and the linkage object itself.
2. **Linking and archiving**. Non-leaf components of the LG, that is, libraries and executable files, are created as a result of linking and archiving processes. To construct a correct LG, each of these objects should be uniquely identified (to avoid ambiguity at the nodes of the LG) and should contain information about the other linkage objects that comprise it (libraries and object files), taking into account their order on the linker command line as well as data on how an existing library was modified. It is quite convenient to identify layout objects using two parameters: the path to the object and its contents hash. Using just one parameter is often insufficient, as some unmonitored commands can modify it. Examples of utilities include `ld`, `ar`, `ld.gold`, `llvm-ar`.
3. **Events that change the linkage object identifier**. During the original build actions may be performed on linkage objects that change their path or contents, necessitating the need to track and process such events. This updates the object identifiers, preventing the loss of linkage data later. When working with dynamic and static libraries and executable files, strip [33] and ranlib [34] are often used as similar utilities. Their focus on linkage objects necessitates their separate processing.

All the above-mentioned processed events with collected data on the layout objects are recorded in the build log, which is one of the results of the build interception necessary for building the LG.

The LG allows us to determine for each two modules whether they are in the same component, and therefore whether procedures from one can call procedures from the other. Unfortunately, the LG may not be sufficient, for example, in cases where modules are added dynamically. For such situations we use the following heuristics:

1. **Using the "single candidate" heuristic**. If there is a call to the function named `foo` and this function is defined only once in the program, then this is most likely the desired function.
2. **Using "path-based" heuristic**. Typically, modules that can call each other share a common path in the source code structure. For calls not resolved in the previous step candidates with the most similar common path are searched for.

JVM languages and Go separate source files into packages. Source files in a single package are assumed to be linked together. These languages additionally apply the **"package specific"** heuristic, which means functions from packages with the same fully qualified name can call each other. In the current implementation this heuristic replaces linkage information collection. Note that in general even for JVM languages and Go it is better to collect linkage information. It is possible for a build script to compile multiple programs that contain packages with the same names (one example is a Android OS build).

To construct the call graph, modules with an IR are read. For each module, an intra-module call graph (local call graph) is constructed, in which nodes can be of two types:

1. module procedures,
2. names of procedures located in other modules.

Then, a global call graph for the entire program is created from the local call graphs and the LG. In the first step, all the local call graphs are merged. The result is a global call graph containing only intra-module calls. For each unresolved call the following actions are performed:

1. A search for candidates for invocation is performed based on the LG so that the called module and the caller are located in the same component.
2. If the previous method was not successful, then the single candidate heuristic is used followed by the path-based heuristic.

5. Devirtualization

As mentioned earlier many languages provide facilities to use virtual calls. Programs in some languages like C utilize this functionality less often, but others like Java, Go and especially Python heavily rely on this mechanism. Performing some sort of analysis is extremely important for the latter. Traditionally analysis of virtual calls is considered as a *devirtualization* problem, i.e., attempting to replace virtual call with a static call when having a single candidate. When constructing a call graph for program analysis we may be interested in more than that. For example, when discovering dependencies all possible candidates for all virtual calls need to be discovered, not just the ones with a single candidate. Just like other tools Svace implements simpler virtual call resolution algorithms relying on type information (namely CHA). However, from our experience these algorithms lack precision needed for accurate call graph reconstruction.

Let us consider the sample code from Fig. 1. Any algorithm that uses type information such as CHA, RTA or VTA will fail here. Both `consumer` methods are identical and thus CHA will report two candidates for invocations of `f()`. RTA and VTA use this information as a base for their analysis and would not be able to properly narrow initial result [7].

Another problem with type-based analysis is its limited ability to resolve function pointer and closure invocations. To address these shortcomings using pointer analysis [35-38] as a base may seem attractive at first. However, when analyzing large programs performance problems arise. Analyzing data-flow of entire program without any regard for type information is costly. And even then, simplest type-based algorithms may produce better results [38].

Our approach to virtual call resolution differs from any of those discussed earlier. First, functions are analyzed independently to produce information about type aliases and external dependencies in each. Information about created function pointers, closures or instantiated objects is used to mark variables as direct aliases of certain types. Variables that depend on data-flow outside of function (such as function parameters, global values etc.) are marked as transitive aliases and saved for second step. Then an iterative algorithm manipulates a dependency graph of function summaries to resolve global interprocedural dataflow. Summaries are processed to see which transitive aliases are updated by this summary and then the algorithm proceeds to update function summaries affected and so on until algorithm eventually converges [39].

This iterative approach takes into account various type information unlike traditional pointer analyses but also does not require starting call graph approximation unlike RTA. This means that in example presented earlier the algorithm can correctly determine which method is invoked in each consumer method. In short, this algorithm can be easily used for:

- C/C++ and Go, if a call graph approximation is available beforehand.
- JVM-languages, where only static calls are resolved.
- Python, as it constructs the call graph from the bottom up.

```
interface I {
    void f();
}
class A implements I { ... }
class B implements I { ... }
void consumer0(I i) {
    i.f();
}
void consumer1(I i) {
    i.f();
}
void example() {
    consumer0(new A());
    consumer1(new B());
}
```

Fig. 1. Advanced devirtualization example.

6. Special cases

Compiling programs in different languages needs specific features and tools. Besides this, various compiler IRs are used. Therefore, the exact algorithm for constructing the call graph in Svace depends on the supported language.

6.1. C/C++

As mentioned above, the C/C++ implementation is the most complete; currently, only for these languages LG is constructed and calls to utilities that affect program linking are intercepted. Another feature of C/C++ analysis is using LLVM IR as input. It is a bitmapped format. At a low level, each file is divided into blocks, which can contain nested blocks. The code for each function is located in a separate block. Additionally, all similar information (constants, metadata, code) is located in separate blocks. The length of each block is written at the beginning allowing blocks to be skipped if they are not of interest.

A significant part of LLVM IR is metadata with debugging information. Since this information is not required for constructing a call graph, a fast LLVM walk was implemented to construct the call graph. To achieve this, only blocks related to the instructions of individual functions were read, which significantly accelerated call graph construction. For example, reading all modules of the Tizen 2.3 project takes 78.4 seconds, while reading modules just for call graph construction takes 14.2 seconds (on the machine mentioned in Section 8).

Another feature of C/C++ is that in some cases code is compiled for different targets (32-bit and 64-bit platforms, x86_64, ARM, RISC-V etc.). Modules for different targets are not designed to be linked together. Therefore, they are analyzed independently, which reduces memory consumption by analyzing each target sequentially.

6.2. JVM languages

For JVM languages (Java, Kotlin and Scala) we build a call graph by compiling a program into JVM bytecode [40], then traversing it and inspecting invocation instructions [41-43]. Dependencies from JAR libraries are also included in the call graph, namely, bytecode of used libraries is collected by analyzing compiler invocations and searching for class-path/module-path in them.

Using JVM bytecode as IR has several advantages:

- Support of multiple languages (Java, Kotlin and Scala) becomes simpler, since IR is analyzed uniformly.

- Cross-language call graphs can be constructed. For example, if a Kotlin method calls a Java method, then compiler would generate this call in a bytecode, thus it will be handled by CodeSkeleton.
- JVM bytecode contains both explicit and implicit calls that describe program semantics completely (for example, invocations of default super-class constructors are generated by the compiler but are not explicitly present in the source code).

On the other hand, there are some limitations to our bytecode-based approach. Since JVM bytecode is a low-level representation mostly suitable for Java, language-specific information of other languages may be lost, such as:

- Rich type information from Scala programs, which can result in call graph precision loss [44].
- Inline-methods from Kotlin. Since inlining in Kotlin can't be completely disabled, there are no calls to these methods in bytecode, thus a bytecode-based call graph won't contain them.

6.3. Go

For Go programs we construct a call graph not only for an entire project but also for dependencies specified in the `go.mod` file. The IR used for analysis and call graph construction is generated during build stage [45] using a modified version of `ssadump` [46] from `go-tools` [19]. `ssadump` is a utility developed by the Go team for generating the IR of Go programs in Static Single Assignment (SSA) form. We have modified this tool to generate the SSA-based IR and construct a local call graph for a given package. The local graphs are then refined during the construction of the global call graph for the entire project. Function linking is performed based on import path [49], exactly as in the original Go compiler build process.

When analyzing Go programs, one must account for several language-specific features that complicate the construction of a complete and precise call graph:

- Interfaces [47] (e.g., `io.Reader`), as we need to find types implementing them.
- Reflection [48] (e.g., `reflect.Value.Call`).
- First-class functions (e.g., `var f = someFunc; f()`).

6.4. Python

For Python programs we don't use any external tools to construct call graph approximation, nor do we use any heuristics for call resolution. As mentioned earlier all work is handled by our dataflow-aware devirtualization algorithm. This decision is motivated by underlying Python semantics. Python does not have static calls like C, C++, Go and JVM languages. Calls are always resolved dynamically, which is illustrated by the code from Fig. 2.

```
def strange():
    print("Strange function")
    global strange
    strange = lambda: print("Other func")
strange()
strange()
```

Fig. 2. Python function redefinition example.

In this code the first invocation of `strange` calls the expected function, however second calls the lambda that is created inside the function itself. This technique and many of its variation where function definition is selected among many options or replaced during runtime often in completely different place (that is often called "monkey patching") is not some niche trick. Our experience

suggests that many sufficiently complex Python libraries use it to define functionality dependent on host operating system, configuration etc.

This means that any function in Python code can potentially be “swapped out” in different part of code. Our devirtualization algorithm can correctly identify that invocation of `strange` in listing 2 can result in call to either candidates, but methods that purely use names of function declaration will fail in this or similar scenarios.

7. CodeSkeleton

As a result of this work, CodeSkeleton, a standalone tool, was developed. It is designed to construct call graphs for programs written in C/C++, Java, Kotlin, Scala, Go, and Python. The tool takes into account the specific call graph construction features mentioned earlier. Its key advantage is not only its support for multiple programming languages but also the ability to automatically intercept the build process, a feature lacking in most existing solutions. This makes CodeSkeleton suitable for the analysis of large software systems with complex build scripts.

When creating CodeSkeleton we wanted to obtain the same call graph as Svace. The best solution would be to use the same code in the same way. Separating the libraries for call graph construction would not fully solve the problem, as the build code would be identical, but it could be used differently. This would result in the accumulation of differences that are difficult to debug, and moreover, supporting two different library usages would consume unnecessary resources.

For these reasons, it was decided to separate the analysis code into a separate component, which required redesigning the Svace architecture. During the process the analysis module was split into two parts: a runner and an analyzer. The runner component is responsible for reading the IR and constructing the call graph. It then calls the analysis plugin, which can analyze the program and is dynamically loaded. If this component is missing, the analysis is not performed. This ensures complete unification of the call graph usage both within and outside of the analyzer. Thus, CodeSkeleton is essentially Svace without the analysis and some auxiliary components. It works for the same languages for which Svace can perform interprocedural analysis. Adding support for C# and Visual Basic.NET, which are implemented in a separate engine, is planned for early 2026.

The solution architecture is shown in Fig. 3. The implementation utilized a Java module system, which allows loading of plugins (modules) with compiler checking.

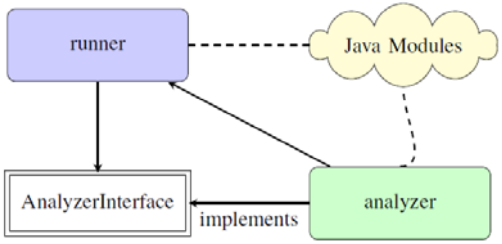

Fig. 3. Architecture schema.

8. Experiments

To compare call graph construction tools, we have selected two projects for each language (C/C++, JVM-languages, Go, and Python): one under 100 KLOC and one over 500 KLOC. For each project we constructed a call graph and measured its construction time, as well as the number of edges in the resulting call graph (see Table 1, columns A and E, respectively). These two metrics were chosen to demonstrate the performance of existing tools and compare them with CodeSkeleton. For more qualitative comparison, it is necessary to verify the precision and completeness of call graph

construction, but unfortunately, we have not found a suitable benchmark. This remains a task for our future work.

All timing measurements were obtained from a server with an Intel(R) Core(TM) i7-14700, 32 GB RAM, and an SSD. Multigraphs, where multiple calls to a procedure bar from a procedure foo are represented as multiple edges, were pre-transformed to count only unique edges in the call graph. This transformation was required because some tools, such as `go-callgraph` and `SootUp`, contain such duplicate edges.

Table 1. Analysis time (T, seconds) and number of edges (E, thousands).

Lang	Project (https://github.com/*)	Size, KLOC	Code Skeleton		SootUp		go-callgraph	
			A	E	A	E	A	E
C/C++	brazilofmux/tinymux [50] (tag: Build-2.6.5.28)	1537	2	8.6				
C/C++	xbmc/xbmc [51] (commit: c6f500ac)	1304	53	552.8				
Java	apache/commons-lang [52] (tag: rel/commons-lang-3.19.0)	926	9	21.7	7	18.7		
Kotlin	JetBrains/kotlin [12] (tag: v1.9.22)	747	152	915.4	434	1300		
Go	pdfcpu/pdfcpu [53] (tag: v0.11.1)	709	18	13.5			1	12.1
Go	cilium/cilium [54] (tag: 1.18.3)	372	120	56.9			68	144.7
Python	pallets/flask [55] (tag: 3.1.2)	182	3	0.7				
Python	pytorch/pytorch [56] (tag: v2.9.0)	180	71	137.2				

For `SootUp` we used RTA algorithm for call graph construction. We excluded functions from standard library, external libraries, abstract and native methods from our and `SootUp`’s call graphs. Also we excluded calls to `"<clinit>"` methods from `SootUp` call graphs since `SootUp` inserts such calls on any access to any class and this is not convenient for comparison. Method calls that start with `"access$"` prefix were also excluded from `SootUp` call graph, since Java compiler in Svace transforms such synthetic invocations into field access instructions.

The `go-callgraph` tool was run with the `-algo=vta` option. From the resulting call graphs, we have filtered out all edges originating from the standard library or external dependencies, retaining only calls within the project itself, i.e. edges where the caller belongs to `github.com/pdfcpu/pdfcpu` or `github.com/cilium/cilium`, respectively. The same filtering procedure was applied to the output of our own tool.

The analysis of call graph construction time reveals that all tools complete within seconds on small-scale projects. For larger projects, CodeSkeleton demonstrates a performance advantage, being several times faster than `SootUp` and comparable to `go-callgraph`.

Regarding the edge count of the constructed call graphs, CodeSkeleton produces a similar number of edges to the other tools on small projects. For larger projects, however, `SootUp` and `go-callgraph` generate significantly larger call graphs. This can be attributed to their use of less precise devirtualization algorithms (RTA/VTA), which lead to an overapproximation of possible targets for virtual calls, thereby inflating the number of edges.

9. Conclusion

In this paper we have discussed the task of constructing call graphs for programs written in different programming languages including C, C++, Java, Kotlin, Scala, Go, and Python. We have analyzed the capabilities of existing tools and frameworks. Our review demonstrated that while there are numerous open-source solutions for individual languages, most of them lack automation of build interception and cannot be easily applied to large software systems, particularly for C/C++ and Python.

We proposed and implemented CodeSkeleton, a unified tool, which supports call graph construction for mentioned programming languages. CodeSkeleton incorporates automatic build interception, linkage reconstruction, and language-specific analysis to construct a call graph. Performed experiments confirmed that our approach is applicable to real-world projects of varying size and complexity. Our tool's more precise devirtualization algorithm produces call graphs with fewer edges than those generated by tools based on CHA. CHA's inherent imprecision leads to a substantial overapproximation of potential targets for virtual calls, inflating edge count.

Future work includes extending the tool to additional programming languages, such as C# and Visual Basic .NET, which are supported by Svace, improving precision of devirtualization algorithms, and developing benchmarks for evaluating the completeness and accuracy of call graph construction.

References

[1]. Polyakov S.A., Borodin A.E. Deadlock Detection using Static Analysis. In Proceedings of the Institute for System Programming of the RAS 32.5 (2020), pp. 21-34.
[2]. Ding B. et al. MEA2: A Lightweight Field- Sensitive Escape Analysis with Points-to Calculation for Golang. In Proceedings of the ACM on Programming Languages 8.OOPSLA2 (2024), pp. 1362-1389.
[3]. Visual Studio Code. Available at: https://code.visualstudio.com/, accessed 10.04.2025.
[4]. IntelliJ IDEA. Available at: https://www.jetbrains.com/idea/, accessed 11.07.2025.
[5]. Ivannikov V.P. et al. Static analyzer Svace for finding defects in a source program code. In Programming and Computer Software 40.5 (2014), pp. 265-275.
[6]. Borodin A.E., Belevantsev A.A. A static analysis tool Svace as a collection of analyzers with various complexity levels. In Proceedings of the Institute for System Programming of the RAS 27.6 (2015), pp. 111-134.
[7]. Bacon D. F., Sweeney P. F. Fast static analysis of C++ virtual function calls. In Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA '96. San Jose, California, USA: Association for Computing Machinery, 1996, pp. 324-341. ISBN: 089791788X. DOI: 10.1145/236337.236371.
[8]. Sõnajalg S. Program analysis techniques for method call devirtualization in object-oriented languages. 2009. Available at: http://www.cs.ut.ee/varmo/seminar/-sem09S/final/s6najalg.pdf, accessed 01.02.2025.
[9]. Anderson P. et al. Flow insensitive points-to sets. In Information and Software Technology 44.13 (2002). Special Issue on Source Code Analysis and Manipulation (SCAM), pp. 743-754. ISSN: 0950-5849. DOI: 10.1016/S0950-5849(02)00105-2.
[10]. Linux. Available at: https://www.linux.org/, accessed 11.07.2025.
[11]. Production-Grade Container Scheduling and Management. Available at: https://github.com/kubernetes/kubernetes, accessed 11.01.2025.
[12]. The Kotlin Programming Language. Available at: https://github.com/JetBrains/kotlin, accessed 11.01.2025.
[13]. The LLVM Compiler Infrastructure project. Available at: https://llvm.org/, accessed 11.07.2025.
[14]. LLVM Optimizer. Available at: https://llvm.org/docs/CommandGuide/opt.html, accessed 11.07.2025.
[15]. Kadiray Karakaya et al. Sootup: A redesign of the soot static analysis framework. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer. 2024, pp. 229-247.
[16]. Watson T.J. Libraries for Analysis, with front ends for Java, Android, and JavaScript, and many common static program analyses. Available at: https://github.com/wala/WALA, accessed 11.01.2025.

[17]. Karim A. et al. A study of call graph construction for jvm-hosted languages. In IEEE transactions on software engineering 47.12 (2019), pp. 2644-2666.
[18]. Cytron R. et al. Efficiently computing static single assignment form and the control dependence graph. In ACM Transactions on Programming Languages and Systems (TOPLAS) 13.4 (1991), pp. 451-490.
[19]. Go Tools. Available at: https://godoc.org/golang.org/x/tools, accessed 09.05.2025.
[20]. Variable Type Analysis (VTA) Algorithm. Available at: https://golang.org/x/tools/go/callgraph/vta, accessed 11.01.2025.
[21]. Package pointer implements Andersen's analysis. Available at: https://pkg.go.dev/golang.org/x/tools/go/pointer, accessed 09.05.2025.
[22]. x/tools/go/pointer: isolate, tag, and delete the pointer analysis. Available at: https://github.com/golang/go/issues/59676, accessed 09.05.2025.
[23]. Go callgraph. Available at: https://godoc.org/golang.org/x/tools, accessed 09.05.2025.
[24]. go-callvis. Available at: https://github.com/ondrajz/go- callvis, accessed 09.05.2025.
[25]. gocyto. Available at: https://github.com/protolambda/gocyto, accessed 09.05.2025.
[26]. Salis V. et al. Pycg: Practical call graph generation in python. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE. 2021, pp. 1646-1657.
[27]. python-call-graph. Available at: https://pypi.org/project/python-call-graph/, accessed 11.07.2025.
[28]. pyan3. Available at: https://pypi.org/project/pyan3/, accessed 11.07.2025.
[29]. CodeQL. Available at: https://codeql.github.com/, accessed 11.01.2025.
[30]. Open-source code analysis platform for C/C++/Java/Binary/Javascript/Python/Kotlin based on code property graphs. Available at: https://github.com/joernio/joern, accessed 11.01.2025.
[31]. Belevantsev A.A., Izbyshev A.O., Zhurikhin D.M. Monitoring program builds for Svace static analyzer. In System administrator 7-8 (2017), pp. 135-139.
[32]. Belevantsev A.A. Multilevel static analysis for improving program quality. In Programming and Computer Software 6 (2017), pp. 3-25.
[33]. strip(1) Linux manual page. Available at: https://man7.org/linux/man-pages/man1/strip.1.html, accessed 11.07.2025.
[34]. ranlib(1) Linux manual page. Available at: https://man7.org/linux/man-pages/man1/ranlib.1.html, accessed 11.07.2025.
[35]. Steensgaard B. Points-to analysis in almost linear time. In Proceedings of the 23rd ACM SIGPLAN/SIGACT symposium on Principles of programming languages. 1996, pp. 32-41.
[36]. Das M. Unification-based pointer analysis with directional assignments. In Acm Sigplan Notices 35.5 (2000), pp. 35-46.
[37]. Cooper K. D. and Kennedy K. Fast interprocedural alias analysis. In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1989, pp. 49-59.
[38]. Whaley J., Lam M.S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation, 2004, pp. 131-144.
[39]. Galustov A.L., Borodin A.E., Belevantsev A.A. Devirtualization for static analysis with low level intermediate representation. In 2022 Ivannikov Ispras Open Conference (ISPRAS). IEEE, 2022, pp. 18-23.
[40]. The Java Virtual Machine Specification. Java SE 21 Edition. Available at: https://docs.oracle.com/javase/specs/jvms/se21/html, accessed 11.01.2025.
[41]. Merkulov A.P., Polyakov S.A., Belevantsev A.A. Supporting Java programming in the Svace static analyzer. In Proceedings of the Institute for System Programming of the RAS 29.3 (2017), pp. 57-74.
[42]. Afanasyev V.O. et al. Kotlin from the perspective of a static analyzer developer. In Proceedings of the Institute for System Programming of the RAS 33.6 (2021), pp. 67-82.
[43]. Afanasyev V.O., Borodin A.E., Belevantsev A.A. Static Analysis for Scala. In Proceedings of the Institute for System Programming of the RAS 36.3 (2024), pp. 9-20.
[44]. Karim A. et al. Constructing call graphs of Scala programs. In European Conference on Object-Oriented Programming. Springer. 2014, pp. 54-79.
[45]. Dvortsova V. et al. Static Analysis for Go: Build Interception. In 2023 Ivannikov Ispras Open Conference (ISPRAS). IEEE. 2023, pp. 52-57.
[46]. Ssadump. Available at: https://pkg.go.dev/golang.org/x/tools/cmd/ssadump, accessed 10.05.2025.
[47]. Go interface types. Available at: https://go.dev/ref/spec#Interface_types, accessed 11.01.2025.
[48]. Package reflect. Available at: https://pkg.go.dev/reflect#sectiondocumentation, accessed 11.01.2025.
[49]. Go import path syntax. Available at: https://pkg.go.dev/cmd/go#hdr-Import_path_syntax, accessed 11.01.2025.

[50]. TinyMUX. Available at: https://tinymux.org, accessed 11.07.2025.
[51]. Kodi. Available at: https://kodi.tv/, accessed 11.07.2025.
[52]. Apache Commons Lang. Available at: https://github.com/apache/commons-lang, accessed 11.01.2025.
[53]. A PDF processor written in Go. Available at: https://github.com/pdfcpu/pdfcpu, accessed 10.04.2025.
[54]. eBPF-based Networking, Security, and Observability. Available at: https://github.com/cilium/cilium, accessed 10.04.2025.
[55]. The Python micro framework for building web applications. Available at: https://github.com/pallets/flask, accessed 10.04.2025.
[56]. Tensors and Dynamic neural networks in Python with strong GPU acceleration. Available at: https://github.com/pytorch/pytorch, accessed 10.04.2025.

Информация об авторах / Information about authors

Артем ФАРТЫГИН – магистрант, стажёр-исследователь ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Artem FARTYGIN – master student, researcher at ISP RAS. Research interests: static analysis for finding errors in source code.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – Cand. Sci. (Phys.-Math.), senior researcher. His research interests: static analysis for finding errors in source code.

Варвара Викторовна ДВОРЦОВА – сотрудник ИСП РАН, аспирант ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ, анализ Golang.

Varvara Viktorovna DVORTSOVA – ISP RAS researcher, postgraduate student at ISP RAS. Her research interests: compiler technologies, static analysis, Golang analysis.

Виталий Олегович АФАНАСЬЕВ – аспирант, младший научный сотрудник ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ, JVM-языки.

Vitaly Olegovich AFANASYEV – ISP RAS researcher and postgraduate. Research interests: compiler technologies, static analysis, JVM languages.

Артемий Львович ГАЛУСТОВ – аспирант, младший научный сотрудник ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Artemiy Lvovich GALUSTOV – ISP RAS researcher and postgraduate. Research interests: static analysis for finding errors in source code.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, член-корреспондент РАН, ведущий научный сотрудник ИСП РАН, профессор МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr. Sci (Phys.-Math.), Prof., Russian Academy of Sciences Corresponding Member, Leading Researcher at ISP RAS, Professor at MSU. Research interests: static analysis, program optimization, parallel programming.