Devirtualization for static analysis with low level intermediate representation

Artemiy Galustov*†, Alexey Borodin*, and Andrey Belevantsev*†
* *Ivannikov Institute for System Programming of the RAS*
† *Lomonosov Moscow State University*
Moscow, Russia
E-mail: {artemiy.galustov, alexey.borodin, abel}@ispras.ru

***Abstract*—We propose a points-to analysis that can recover targets for function pointer calls, virtual calls and method calls for using in a static analysis. We use a flow-insensitive analysis, and the analysis results are intended for flow- and path-sensitive analysis which can improve the initial analysis precision within a single function. We implemented the proposed approach in a static analyzer for finding errors in C, C++, Go, Java and Kotlin programs. The devirtualization algorithm is fast enough and spends less than 6% of the total analysis time. It can work for projects like Tizen 7 with 27.5 MLoc of source code.**

***Index Terms*—static analysis; pointer analysis; devirtualization; Java; Kotlin; C; C++; Go; SVACE; JVM; llvm.**

I. Introduction

It is almost impossible to imagine the modern programming without some form of virtual calls. Even some of the earliest computer languages featured virtual call mechanisms via function pointers, and with advent of object-oriented programming polymorphic classes' behavior promoted the virtual call mechanism from an occasionally useful tool to the basis of software development. Therefore performing software static analysis is heavily dependant on solving the devirtualization problem. The solutions available in compilers or static analysis suites are many and varying, and they are not always suitable to the researchers' needs.

The most general solutions are provided by a points-to analysis [1]. This analysis seeks to answer whether some pointer p can or can not point to some place in memory q. Many of these algorithms naturally support function pointers and can handle C/C++ code, however handling languages such as Java and Go, where calls are not resolved using function pointers and virtual tables, will require instead tracking methods and binding them to object variables separately. One of the most well known algorithms is Bjarne Steensgaard's points-to analysis [2]. This is a flow-insensitive, interprocedural algorithm that works in $O(N \cdot \alpha(N, N))$ where N is the program length and $\alpha(m, n)$ is the inverse Ackermann function [3]. The algorithm uses an equivalence based approach representing all program variables in a disjoint set data structure [3] and joining them based on a set of rules for different operations. This ensures good complexity but sometimes produces poor results [1]. A more granular approach is using binary decision diagrams as presented in the work of John Whaley et al. [4]. This algorithm has both context-sensitive and insensitive versions and is inclusion based. The approach provides better results and is designed to work with OOP-heavy languages like Java. It can handle relatively big programs as authors tested it on code bases up to 300K lines of code.

Despite all benefits points-to analysis isn't well suited for devirtualization. Not only do some researchers point out that point-to analysis can perform worse than type analysis [4], but the general expediency of such approach for devirtualization is questionable. A points-to algorithm analyzes all dataflow in a program and disregards information about types and related virtual calls. Techniques such as Class Hierarchy Analysis [5] can produce coarse yet meaningful results without analyzing any program instructions. Simple extensions such as Rapid Type Analysis [5] can produce even better results when coupled with a very simple intraprocedural dataflow analysis.

All of the discussed approaches have drawbacks in precision, performance, or both. Moreover, existing papers focus on devirtualization for one language at a time. Because SVACE is expected to handle large projects (see VIII) in different languages, presented approaches could not be applied as is. In this paper we present the devirtualization approach for an intermediate representation (IR) that supports multiple languages. The described algorithms are implemented in SVACE — static analysis for finding errors in C, C++, Java, Kotlin, Go programs [6–8].

II. Intermediate representation for different languages

SVACE uses the following intermediate representations as an analyss input:

1) LLVM bitcode for C, C++
2) JVM bytecode for Java, Kotlin
3) `ssadump` tool output for Go

All these representations are then transformed into the unified SVACE IR. This format is used by SVACE internally and emulates most of memory model features of the languages being analyzed. Similar to the LLVM IR, our intermediate representation is SSA-based and does not include *address of* operator.

SVACE has some common instructions for operations present in all languages:

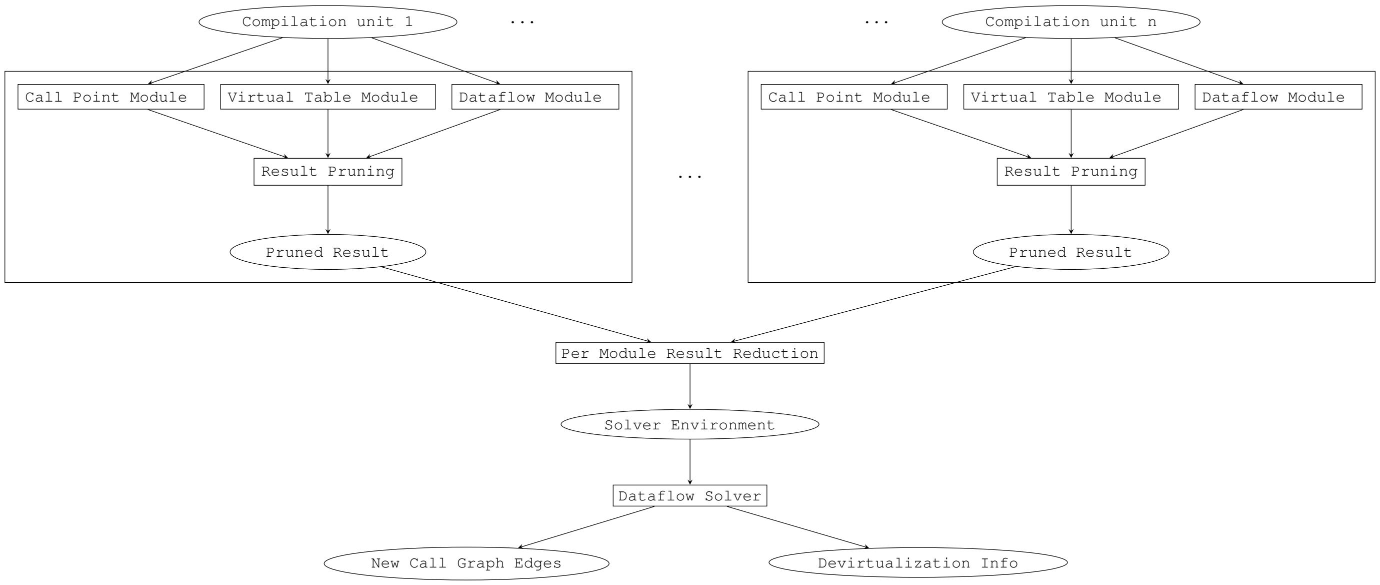

Figure 1: Devirtualization algorithm steps

$a = b$	ssa assignment
$a = *b$	dereference
$a = b.field$	object field read
$*a = b$	memory store
$a.field = b$	object field write
$a = cast\ b$	cast operation reinterpreting value of b
$a = func(...)$	direct call of $func$ with arguments
$return\ a$	return a function value
$a = (*ptr)(...)$	function pointer call with arguments
$a = ptr.func(...)$	method call for object with arguments

The first five instructions are common for all languages analyzed by SVACE. Function pointer calls can only be found in LLVM and Go languages. Method calls are specific to Java and Go that do not rely on virtual tables unlike C++.

III. APPROACH TO DEVIRTUALIZATION IN SVACE

SVACE performs analysis in three *phases*:

1) Building a call graph. During this phase we construct a graph out of all direct function calls.
2) Preliminary phase performing lightweight analysis on all compilation units to gather additional information about analyzed code before the main phase.
3) Main Phase, a summary-based function analysis that uses the call graph information.

To implement devirtualization we have extended the preliminary phase. We use the data gathered in the preliminary phase to update the call graph before the start of the main phase, and the data is also available in the main phase in a special data structure.

The call graph must contain all edges to functions which may be called. This is required to build a summary of each called function before the function analysis on the main phase. We also permit a situation when the graph contains edges to functions that are never called. This is still useful information for the analysis because the static analysis intent is to process all possibly valid program states whereas a compiler is only interested in places with single resolution [9, 10]. However, the result size is still a concern from a performance standpoint. SVACE will disregard devirtualization results for calls with abnormally many candidates. From our experience more than 10 candidates per call do not produce any useful warnings but slow down analysis significantly.

Because SVACE targets different languages with different memory models and different language structures related to virtual calls, the devirtualization algorithm needs to unify available data into a single language-agnostic format. Moreover, SVACE is rapidly developing and adding support for new languages, so the devirtualization algorithm must be easily adaptable to all of them. To solve these problems and to fit into the existing tool infrastructure a modular approach illustrated in the fig. 1 is used.

Each compilation unit is independently analyzed by three separate modules:

- *Dataflow Module* analyzes functions and builds a *summary*. During this step each function is processed independently. The dataflow analysis uses the notion of *aliases*. Each variable can alias some concrete types, e.g. a class, a structure, a function pointer, or can be a *transitive alias*. A *transitive alias* is a reference to the data outside of a function: parameters, return values and class fields, but transitive aliases do not reference data inside the same function. Each variable in the function summary is aliased in such a way that it is independent from other variables. Variables used as return values, call parameters or field values are marked as useful to avoid deletion later. For this purpose we use a set of instruction rules that we describe with the following notation:

$ALIAS(x)$	set of aliases for variable x
$USEFUL$	set of useful variables in a function
$transitive(x)$	transitive alias for x
$concrete(type)$	concrete alias for $type$
$build(type)$	creating an alias

It is also important if some instruction argument is a local or a global symbol. Local symbols can only be function variables. A global symbol can be either a variable in global scope, a field of class or a function argument. This types of symbols are treated equally by dataflow propagation algorithm treats them equally and uses the same mechanism is used to update their values. Some rules can not operate on global symbols, so the below rules are non-exhaustive.

$$\frac{a = build(type)}{ALIAS(a) \leftarrow concrete(type)}$$

$$\frac{\dfrac{a \text{ is local, } b \text{ is local}}{\begin{array}{l} a = b \\ a = cast\ b \\ *a = b \end{array}}}{ALIAS(a) \leftarrow ALIAS(b)}$$

$$\frac{\dfrac{a \text{ is global}}{\begin{array}{l} *a = b \\ c.field = b \\ return\ b \end{array}}}{USEFUL \leftarrow b}$$

$$\frac{\dfrac{b \text{ is global}}{\begin{array}{l} a = b \\ a = cast\ b \\ *a = b \end{array}}}{\dfrac{\begin{array}{l} ALIAS(a) \leftarrow transitive(b) \\ a = b.field \end{array}}{ALIAS(a) \leftarrow transitive(field)}}$$

$$\frac{\begin{array}{l} a = func(b_1, b_2, ...b_n) \\ a = (*ptr)(b_1, b_2, ...b_n) \\ a = ptr.func(b_1, b_2, ...b_n) \end{array}}{\begin{array}{l} ALIAS(a) \leftarrow transitive(ret\ of\ func) \\ USEFUL \leftarrow ptr, b_1, b_2, \ldots, b_n \end{array}}$$

The rules corresponding to *build* are specific to each language and are described later in the Sections IV, V, VI. In general they are created at a place of object instantiation or are produced by specific instructions that create function pointers.

- *Call Point Module* finds all instructions that result in a virtual call and creates a Call Point object representing such a code location. Call Point objects may contain any language-specific information internally. However, their interface only provides information about the variable containing an object or a function pointer and can resolve a virtual call given the type aliasing information.
- *Virtual Table Module* reads all top level data and builds the tables connecting types with their methods. This module processes virtual tables for C++ (see Section IV) and analyzes method definition for JVM and Go (see Sections V, VI).

When the modules finish processing a compilation unit, their joint information is used to prune redundant information from the Dataflow Module. All aliases for types not present in the Virtual Table Module can be removed. Using set of useful variables obtained by the Dataflow Module and the Call Point Module information, any variables that are not used to resolve virtual calls, to update field values or are not used as parameters or return values can be pruned. Because transitive aliases, by definition, only alias these entities, but not other variables inside a function, removing such variables does not change the information describing a function.

The pruned module info can not be used yet to perform analysis on an entire program. Before starting a solver, all pruned results are reduced into the single Solver Environment. During reduction step the information from individual compilation units is moved into a single container, and language-specific updates are done for Virtual Table Modules (see Sections IV, V, VI).

With all information extracted from compilation units and combined into one language-agnostic representation, the final algorithm step takes place.

Dataflow Propagation Algorithm

```
Require: $queue$ = [ all function in environment ]
 1: while $queue$ is not empty do
 2:     $f$ = next from $queue$
 3:     if not $updates(function)$ has new concrete aliases then
 4:         continue
 5:     end if
 6:     $newUpdates = []$
 7:     for each updated $b$ in $updates(f)$ do
 8:         for each value $a \leftarrow transitive(b)$ do
 9:             $a \leftarrow updates(f)$ for $b$
10:             $newUpdates \leftarrow a$
11:         end for
12:         if resolves virtual call $a = (*b)(...)$ or $a = b.func(...)$ then
13:             add an edge to the call graph
14:             $updates(f) \leftarrow a$
15:         end if
16:     end for
17:     for each $f'$ adjacent to $f$ do
18:         $updates(f') \leftarrow newUpdates$
19:         add $f'$ to $queue$
20:     end for
21: end while
```

The Dataflow Solver processes one function at a time (within a separate environment) and tries to propagate concrete type aliases to all variables that have transitive aliases. These transitive aliases come from other functions that either call this function or are being called by it. Therefore all dataflow information can be propagated along call graph edges that are being dynamically reconstructed by the Dataflow Solver only using the summary built by the Dataflow Module earlier. This can be efficiently done using an iterative approach similar to [11], except that interprocedural phases are replaced by updates of function summaries, which take only fraction of the original time. To process a single function summary update, a solver checks which arguments and return values are updated and then updates variables as needed transitively aliasing them. If nothing is to be updated then the algorithm skips this summary. Otherwise the algorithm checks if the updated concrete aliases can resolve some virtual calls and adds all edges representing resolved virtual calls. This constitutes the dynamic reconstruction of the call graph edges. Finally all functions adjacent to the processed one are marked as changed and are queued for update. Unlike [11] there is no need to check if an alias has or has not been propagated through the edge, because the summary analysis will not commence if it was propagated earlier. It is obvious that concrete alias sets propagated via this method are finite and thus form a half lattice, which is required for the algorithm to terminate [12]. What makes this approach different from classic iterative algorithms is the call graph edge reconstruction. One may think that adding new edges may interfere with the iteration order, but it is not the case. On one hand, edges are always added to the last analyzed vertex. On the other hand, the number of edges in the call graph is finite. This means that at some (finite) iteration each vertex will have the full set of edges to continue propagation through. When the last edge is added, the algorithm becomes equivalent to the one that does not add new edges.

IV. Devirtualization for C function pointers and C++ virtual method calls

One of the most important targets for Svace are C and C++ programs. They feature two kinds of virtual call mechanisms: function pointers and classes with virtual methods.

Building virtual tables for C++ is straightforward. Svace intermediate representation contains all global variable declarations and C++ virtual tables are just one type of these declarations. An array of function pointers is supplied and parsed to get signatures of the functions associated with a single class. All function pointers inherited or declared by the class itself are stored in one place and no further work to find signatures is needed. Finally the Virtual Table Module reads definition of all functions and builds a map from a function to a signature. The map is used at the reduction step of building environment to pair signatures acquired from the virtual table definitions with functions themselves. This ensures that if a function body and a class body are declared in different compilation units then the function is still properly represented in the virtual table.

Concrete type aliases in C/C++ code can be created in two ways. For function pointer creation points in the C/C++ code such as **void** $(*\text{ptr})(...)$ = &foo; we emit the $a = cast\ b$ instruction is emitted in the IR. These instructions contain function signatures and allow pairing function pointers and their definitions the same way as for virtual tables. Class instantiations are represented via direct function calls $func(a, b, ...)$ with a special flag. Constructor signatures allow linking function calls with class type and creating the concrete alias for a C++ class type.

V. Devirtualization for JVM-based languages

All JVM-based languages from classic object-oriented Java [13] and Kotlin [14] to functional Scala [15] share the same platform and are compiled to the same bytecode that was initially created for the Java language [16, 17].

In contrast to C++ JVM does not support function pointer calls and the only virtual call mechanism is via virtual methods. Building virtual tables for JVM languages is different to C++. Java does not have virtual tables for each class. Some methods are described inside class declaration but some will be inherited from superclasses. To find all class methods the Virtual Table Module for JVM collects all virtual method declarations and all superclasses for each class. During reduction step of the building environment each virtual table for the given class is extended with all methods that are not overridden. Unlike C++ all methods are declared in the same compilation unit as the class they belong to, and no additional work is needed to connect class to its methods.

A call point of a virtual method in a JVM program is one of several $invoke*$ instructions [16] represented by a single $ptr.method(...)$ instruction in the Svace intermediate representation. Our devirtualization analysis does not differentiate invocation methods as well and only stores information about an object variable and a method signature.

Just like for C++, we have a special direct call instruction flag for a JVM constructor call that can be used to create concrete aliases for a JVM class type.

VI. Devirtualization for Go

Like C++, Go has function pointers and structure methods that can produce virtual calls. But unlike C++, Go does not feature all OOP constructs. Go does not feature inheritance, replacing it with *embedding* mechanism that does not produce polymorphic classes [18]. The only option to create polymorphic behavior in structures is to use interfaces, which are implemented by all matching structures implicitly (also known as duck typing). All this properties of the Go language make a difference in handling method devirtualization compared to C++ and JVM.

Unlike C++ and JVM, the special type of $a = cast\ b$ instruction is used to create a concrete alias for a Go type. The SSAdump tool for Go has a *make interface* cast type that represents a type change from a Go structure type to a Go interface type. Because structures in Go can not be abstract and can not be inherited, the initial type is the type of the value and can be used to create the concrete type alias.

As Go features duck typing, any structure is a candidate to be used through an interface. This would result in the big

Virtual Method Table and would slow down analysis. One can use interface definitions to remove entries for types that do not match any interface. However, it is much more efficient to remove all types that are never aliased in functions. This would leave only tables for types that are *cast* to some interface and would remove any non-polymorphic Go structures.

Handling method call points is done similarly to JVM and handling function pointers is identical to C++ in SVACE IR.

VII. USING DEVIRTUALIZATION RESULTS

We propose the following scheme for using devirtualization results which allows improving precision of devirtualization analysis.

For function analysis we use symbolic execution with merging analysis states at path joins. We provide the detailed description at [8].

Devirtualization results for every external function and pointer symbol return a set of destination functions. Since the analysis used is flow-insensitive, it may have imprecise results. But the main analysis is flow- and path-sensitive[1]. For pointer symbols the analysis remembers the pointed destination procedures. In case of path joining, this analysis does not use information from devirtualization and joins those sets itself because the devirtualization algorithm cannot provide more precise results.

Moreover, the data flow information is traversed over other pointers, array elements and structure fields. Interprocedural and intermodule analysis is used. All those features allow propagating further the initial points-to information. Listing 1 is an example where the points-to information may be improved on the main analysis phase. The devirtualization analysis computed that ‘p2‘ can point to ‘func1‘ or ‘func2‘, so ‘x‘ may be 10 or 0. Our analysis takes flow into account and improves precision computing that pointer ‘p2‘ may point only to ‘func1‘. As a result, the analysis can figure out that code execution will definitely lead to buffer overflow.

[1]Our current version does not use path-sensitivity for improving devirtualization but it is possible to implement.

```
int func1() {return 10;}
int func2() {return 0;}

typedef int (*Fptr)();
char buf[10];

void foo (Fptr p1, Fptr p2) {
    p2 = p1;

    int x = p2 ();
    buf[x] = 1;//buffer overflow
}

void caller () {
    foo (&func1, &func2);
}
```

Listing 1: Improving devirtualization results

If there is only one destination candidate, then the current instruction will be replaced to the direct call of the candidate function. In case of several candidates our analysis splits paths and processes every candidate separately. After that all paths are merged together.

VIII. RESULTS

To test algorithm and implementation in SVACE we analyzed projects ranging in size from thousands to tenth of millions of lines of code. To demonstrate effectiveness of the presented approach in different usage scenarios, two different machines were used. To analyze larger projects (fig. 2), a server with 16 CPU cores and 256 Gb of RAM was used. To analyze smaller projects (fig. 3), a PC with 4 CPU cores and 16 Gb of RAM was used. The number of resolved virtual calls is calculated after removing calls with abnormally many candidates. Number of virtual call points includes calls to standard library and dependencies, which are not captured by SVACE and thus candidates are not present in intermediate representation. To illustrate the algorithm performance, we show relative ratio of devirtualization to total runtime of all SVACE components. Results demonstrate that our approach

Project	Source Language	Size (MLOC)	Virtual Calls Resolved	Number of Virtaul Call Points	Algorithm Runtime (seconds)	% of Total Analysis Time
Tizen 7	C++	27.5	60491	387561	360	1.5%
Tizen 2.3	C++	6.6	8567	57005	32	<1%
Android 12	Java	33	94291	295793	184.1	3%

Figure 2: Results for large projects.

Project	Source Language	Size (KLOC)	Virtual Calls Resolved	Number of Virtaul Call Points	Algorithm Runtime (seconds)	% of Total Analysis Time
NanoVM OPS	Go	24.1	1010	9250	4.8	<1%
Kodi	C++	879.6	3629	31263	130	1%
Recaf	Java	40.0	551	6440	4.4	<1%

Figure 3: Results for small projects.

works well both for big programs relying on virtual calls heavily as well as for smaller programs that do not rely on virtual calls as much. Moreover, execution time of the devirtualization algorithm becomes significant only on the biggest codebases analyzed by SVACE.

IX. CONCLUSION

This paper introduces a devirtualization algorithm based on type aliasing. This algorithm achieves a couple of important goals. It can support multiple different programming languages. It can analyze projects sized from tens of thousands to tens of millions of lines of code in a reasonable time. And it can tightly integrate with flow- and path-sensitive analysis to provide more precise results.

References

[1]. Derek Rayside et al. "Points-to analysis". In: (Dec. 2005).

[2]. Bjarne Steensgaard. "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.

[3]. Robert Endre Tarjan. *Data Structures and Network Algorithms*. USA: Society for Industrial and Applied Mathematics, 1983. ISBN: 0898711878.

[4]. John Whaley and Monica S Lam. "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.

[5]. Sander Sõnajalg. "Program analysis techniques for method call devirtualization in object-oriented languages". In: 2009.

[6]. V.P. Ivannikov et al. "Static analyzer Svace for finding defects in a source program code". In: *Programming and Computer Software* 40.5 (2014), pp. 265–275.

[7]. AE Borodin et al. "Searching for Taint Vulnerabilities with Svace Static Analysis Tool". In: *Programming and Computer Software* 47.6 (2021), pp. 466–481.

[8]. AE Borodin and IA Dudina. "Intraprocedural Analysis Based on Symbolic Execution for Bug Detection". In: *Programming and Computer Software* 47.8 (2021), pp. 858–865.

[9]. Piotr Padlewski. "Devirtualization in LLVM". In: *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH Companion 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 42–44. ISBN: 9781450355148. DOI: 10.1145/3135932.3135947. URL: https://doi.org/10.1145/3135932.3135947.

[10]. Alexey Pavlovich Merkulov, Sergey Andreyevich Polyakov, and Andrei Andreyevich Belevancev. "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.

[11]. István Forgács. "Double iterative framework for flow-sensitive interprocedural data flow analysis". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 3.1 (1994), pp. 29–55.

[12]. Alfred V Aho et al. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[13]. James Gosling et al. *The Java language specification*. Addison-Wesley Professional, 2021.

[14]. *JetBrains/kotlin. The Kotlin Programming Language*. URL: https://github.com/JetBrains/kotlin (visited on 04/22/2022).

[15]. Martin Odersky et al. "An overview of the Scala programming language". In: (2004).

[16]. Tim Lindholm et al. *The Java Virtual Machine Specification, Java SE 17 Edition*. Addison-Wesley, 2013.

[17]. 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.

[18]. Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.