Modeling the Effects of Global Variables in Data-Flow Analysis for C/C++

Philipp Dominik Schubert*, Florian Sattler†, Fabian Schiebel‡, Ben Hermann§ and Eric Bodden*‡
*Paderborn University, Paderborn, Germany, {philipp.schubert, eric.bodden}@upb.de
†Saarland University, Saarbrücken, Germany, sattlerf@cs.uni-saarland.de
‡Fraunhofer IEM, Paderborn, Germany, fabian.schiebel@iem.fraunhofer.de
§Technische Universität Dortmund, Dortmund, Germany, ben.hermann@cs.tu-dortmund.de

***Abstract*—Global variables make software systems hard to maintain and debug, and break local reasoning. They also impose a non-trivial challenge to static analysis which needs to model its effects to obtain sound analysis results. However, global variable initialization, codes of corresponding constructors and destructors as well as dynamic library code executed during load and unload not only affect control flows but data flows, too. The PhASAR static data-flow analysis framework does not handle these special cases and also does not provide any functionalities to model the effects of globals. Analysis writers are forced to model the desired effects in an ad-hoc manner increasing an analysis' complexity and imposing an additional repetitive task. In this paper, we present the challenges of modeling globals, elaborate on the impact they have on analysis information, and present a suitable model to capture their effects, allowing for an easier development of global-aware static data-flow analyses. We present an implementation of our model within the PhASAR framework and show its usefulness for an IDE-based linear-constant propagation that crucially requires correct modeling of globals for correctness.**

***Index Terms*—static analysis, global variables, C/C++**

I. Introduction

Global variables are best avoided. Not only do they increase the complexity of debugging and maintaining software systems but they also break local reasoning [1]. Global variables are used nonetheless to communicate information when using shared memory parallelism, to implement singletons, and to pass state across multiple functions without parameter passing.

Global variables are not only memory locations that can be accessed at all points in a program but they also come with code for initialization and de-initialization that is executed "before" *and* "after" `main()`—the *actual* program. The situation gets more complex as there are a multitude of different (de-)initializations depending on various conditions. Built-in types such as `int`, `long`, `double` are (de-)initialized differently than user-defined types, for instance. In addition, there is code that is executed whenever a shared library is loaded or unloaded which must be modeled, too.

A client data-flow analysis that verifies some property on a given target program does not only require data-flow information, but, in addition, information from various helper analyses such as callgraph and points-to. Depending on the complexity of the client analysis, it even requires information of additional data-flow analyses. Global (de-)initializers may affect all of these different analysis representations.

Static analyses are typically parameterized with a set of entry points that specify where in the program the analysis must start. Interestingly, the global (de-)initialization code is not explicitly connected with the program's actual entry point(s) such as `main()`. If a user specifies `main()` as an entry point to their analysis in a whole program analysis setup, the analysis still misses all of the global code that is executed "before" and "after" `main()`.

PhASAR [2] currently does not provide framework support for modeling the effects of global variables and associated code. Current existing analysis implementations model globals in an ad-hoc manner or not at all. The effects of global code is modeled by repurposing flow-function implementations making the analysis code unnecessary complex and degrading analysis' performance. It is also unlikely that an ad-hoc handling of globals covers all possible scenarios and leads to sound analysis implementations.

While an unsound handling of globals may be reasonable for analyses such as uninitialized variables, which can safely ignore global variables as those are automatically zero-initialized in C and C++ if a programmer does not provide an initial value, many others crucially depend on a sound and precise handling of globals.

In this work, we thus present a structured overview on how global variables and associated (de-)initialization code are used in C/C++. We explain how these usages are represented in LLVM's [3] intermediate representation (LLVM IR) that is the target of PhASAR [2] and many other analysis tools for C and C++. We elaborate on how to precisely model global effects for sound data-flow analysis and present an extension that we implemented for PhASAR [2] to provide framework support. We show the usefulness of our model and its implementation by presenting a linear-constant propagation that crucially depends on correctly handling globals.

In summary, this paper makes the following contributions:

- A comprehensive overview on the possible usages of global variables and global code in C/C++.
- A model and its open-source implementation in PhASAR [2] that allows static-analysis writers to easily and soundly encode global effects into their analysis.
- A case study and an empirical evaluation that assesses the importance of correctly handling globals [4].

```
1  // An exemplary header file:
2  Global var              extern int i;
3  Global var              static inline int j = 1024;
4                          struct Point {
5  Class member              int a, b;
6  Static class member       static int c;
7  Static class member       static inline int d = 73;
8  Class default ctor        Point();
9  Class ctor                Point(int a, int b);
10 Class dtor                ~Point();
11                         };
12                         Point &getSingletonPoint();
13 // The header's respective implementation file:
14                         #include "overview-globals.h"
15 Global var              int i;
16 Static init             static int k = 42;
17 Global in namespace     namespace ns { int l = 13; }
18 Anonymous namespace     namespace { int m = 9000; }
19 Static class member     int Point::c = 2;
20 Global class var        Point p(42, 13);
21 Class default ctor      Point::Point() : a(0), b(0) {
22                           printf("%d-%d", a, b);
23                         }
24 Class ctor              Point::Point(int a, int b)
25                           :a(a),b(b){printf("%d-%d", a, b);}
26 Class dtor              Point::~Point() { printf("%d", d); }
27                         Point &getSingletonPoint() {
28 Local static              static Point s(11, 22);
29                           return s;
30                         }
31                         __attribute__((constructor))
32 Global ctor             void onLoad() { i = 9001; }
33                         __attribute__((destructor))
34 Global dtor             void onUnload() { i = 0; }
35                         int main() {
36                           Point &q = getSingletonPoint();
37                           return 0;
38                         }
```

Fig. 1: An exemplary header and implementation file.

II. Background and Problem Description

In the following, we first present the various possible usages of global variables in C and C++ and describe their varying semantics depending on the situation they are being used. Then, we explain how the different semantics are represented in LLVM IR. We use these insights to design suitable abstractions that allow for precisely modeling the effects of globals in static data-flow analysis in Section III.

A. Globals in C and C++

We present the different usages of global variables and their associated (de-)initialization code that is executed "before" and "after" the actual main program, respectively, by going through the code of Figure 1 line by line. We annotated the code to improve readability.

a) Built-in typed global variables: Line 2 declares a global variable that can be used across one or more compilation units as long as they contain a declaration of `i`. The variable `i` needs to be defined in exactly one compilation unit in which it is then automatically zero initialized as no explicit initial value is specified (cf. Line 15). The linker refers all users of `i` to this definition. In C and C++, all global variables are initialized with zero at compile time if no value is provided by the user as this does not entail any runtime costs.

Since *C++17*, the C++ standard allows for static inline definitions of global objects, i.e., *functions and variables* in header files (cf. Line 3). Due to the `inline` keyword this does not constitute a violation of the *one definition rule* (ORD). The one definition rule prescribes that non-inline objects (since C++17) and non-inline functions cannot have more than one definition in the entire program. Violations of that rule that span translation units are not required to be diagnosed and result in undefined behavior. Defining objects in header files using the `inline` keyword may produce multiple but equal definitions of the global object and therefore, it does not matter which definition the linker eventually arbitrarily picks and puts into the globals section of the final binary.

Line 6 and Line 7 depict analogous situations for class or struct types. Line 6 declares a global variable `c` that is part of the `Point` type. Similarly to the aforementioned situation, it must be defined in exactly one compilation unit (cf. Line 19). Consequently, the `inline` keyword allows for a definition in a header file without breaking ODR.

Line 16 defines the variable `k` that can be accessed globally but only within the compilation unit it is defined in. Line 18 shows an analogous situation using C++'s anonymous namespaces. The variable `l` is available across multiple compilation units within the namespace `ns`.

b) Class/struct typed global variables: Line 20 defines the global variable `p`. Its constructor runs "at startup" before the C runtime starts the program's execution at `main()`. Its destructor is called before exiting the program at the end of `main()`. An analogous situation is depicted in Line 28. The function `getSingletonPoint()` implements a thread-safe singleton, sometimes referred to as Scott-Meyers-Singleton, of type `Point`. The variable is initialized exactly once when `getSingletonPoint()` is called for the first time. Its destructor is called before the program exits.

c) Global con-/destructors: The definition of `onLoad()` in Line 32 presents a global constructor. Function definitions that are annotated with the `constructor` attribute are executed while the compilation unit that defines these functions is loaded by the loader or the dynamic linker. Functions that are annotated with the `destructor` attribute present global destructors and are executed before the program exists. Line 34 shows an example for such a function. Even though the global constructor/destructor mechanism is currently not part of the C++ standard, it is often used in the context of shared libraries and therefore, should be supported by an analysis. Shared libraries may define several global con- and/or destructors that are executed when a shared library is explicitly loaded by another program using `dlopen()` or `dlsym()` and `dlclose()`, respectively. In combination, this mechanism is used to implement plugins that (de-)register themselves within some other application that uses them.

B. Representation in LLVM IR

All types of global variables presented in the previous section can be found in the LLVM IR as well. Global variables whose access is restricted to the compilation unit or the function they are defined in are marked as `internal global`

. Global built-in data types such as `int`, `char` or `double` are automatically initialized with zero.

Global variables of user-defined types are statically initialized with zero, too. Semantically, all data members of the given type are initialized with zero. Constructors and destructors come into play later.

LLVM provides two special global array variables `llvm.global_ctors` and `llvm.global_dtors` that carry information on the con- and destructors of global variables of user-defined types as well as global con- and destructors. The functions referenced by these arrays will be called in ascending order of priority, i.e., lowest first when the module is (un-)loaded by the loader or the dynamic linker. The order of functions with the same priority is not defined. Programs that introduce dependencies between global variables whose (de-)initialization code has the same priority are invalid.

When our module in Figure 1 is loaded, `onLoad()` is executed. After `onLoad()` has been executed, or before—in our case the priorities are equal, a special function responsible for executing the initialization code of all global, user-defined type variables is executed. Such a function is emitted for each compilation unit, if necessary. The linker handles merging these functions whenever modules are linked. The function itself calls other automatically generated functions each of which is responsible for initializing an individual global variable of a user-defined type. In our example, the function calls `p`'s constructor to correctly initialize it at the program's startup. It also registers `p`'s destructor to be called using the C runtime's `__cxa_atexit()` function. The global variables' constructors are called in order of definition. Their destructors are called in reverse order once the program exits or the module is unloaded. Global destructors such as our `onUnload()` function are registered in the `llvm.global_dtors` variable in an analogous way.

The `Point` singleton, like the other global variables, is zero initialized. Its initialization takes place at the very first call to `getSingletonPoint()`. Depending if a compiler generated guard variable has been set atomically, its constructor is called and its destructor is registered in the C runtime. In case the guard variable is already set, this step is skipped and a reference to the initialized instance is returned directly.

III. Modeling the Effects of Globals

In this section, we present how global variables are currently handled by analysis writers and how one can model the behavior of global variables in a more stringent manner.

A. Status Quo

Current analyses that come with Soot [5] or PhASAR [2] either ignore global variables completely or they repurpose an analysis' flow-function implementations to model their effects.

The current scheme for modeling global variables that is often found in practice is shown in Listing 1. The scheme uses the flow function implementation by adding additional code that is executed once at the very beginning of an analysis. Because this scheme uses a call to the flow function that would normally be used to model the intra-procedural effects of the `Curr` statement, the query for `Curr` must be performed within the global-handling code and its result must be combined with the flow function that describes the effects of the global variables (cf. Line 51). The scheme as is, besides increasing the analysis' complexity, ignores code for (de-)initializing global variables. This handling is also not quite correct as it would not work if non-intra-procedural, i.e., non-*normal*, statements are chosen as entry points. One would therefore need to replicate the global-handling code in the flow functions that handle *call* and *return* flows, too.

```
39 FlowFunctionPtrType getNormalFlowFunction(N Curr, N Succ) {
40   static bool InitGlobals = false;
41   if (!InitGlobals && InitialSeeds.count(Curr)) {
42     InitGlobals = true;
43     std::set<D> ToGenerate;
44     for (auto &Global : getGlobals())
45       if (Global.isConstant())
46         ToGenerate.insert(&Global);
47     auto GlobalsFF = std::make_shared<GenAll<D>>(ToGenerate,
48                                                  ZeroValue);
49     // compute the flow function for the actual statement
50     auto CurrFF = getNormalFlowFunction(Curr, Succ);
51     return std::make_shared<Union<D>>({GlobalsFF, CurrFF});
```

Listing 1: An excerpt of global-variable-handling code using an IFDS [6]/IDE [7] *normal* flow function implementation.

We next describe how the most laborsome parts of modeling globals can be shifted to an analysis framework.

B. Control Flows

To conduct an inter-procedural, i.e., whole program analysis, an inter-procedural control-flow graph (ICFG) is required. An ICFG must be parameterized with one or more entry points $E_0, \ldots, E_n$. In case one wishes to conduct a whole program analysis, `main()` is usually chosen as an entry point. However, by choosing `main()` as an entry point, the ICFG misses lots of control flows that may be crucially important to the client analysis since a lot of functionalities involved in (de-)initialization are executed "before" *and* "after" `main()`.

To produce a sound ICFG that supports whole program analysis, an ICFG algorithm must respect and analyze the global constructors. While analyzing the transitively reachable functions it must also register all functions that are registered to the C runtime using `__cxa_atexit()` and retain their order. Only then an ICFG algorithm may analyze the control flows starting at `main()`. Once the control flows of `main()`—the main program, and transitively reachable functions have been computed, the algorithm must continue analysis in the global destructors and also the destructors that have been previously registered using `__cxa_atexit()` in reverse order until *all* control flows have been analyzed and a complete model of the program under analysis has been constructed.

A schematic overview of an ICFG that respects global variables and global code for a given target program is shown in Figure 2. First, a global-aware ICFG must respect the primary initialization of global variables in order of appearance in the code indicated by the box labeled *I*. Note that

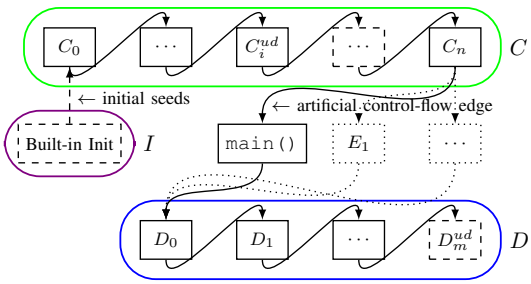

Fig. 2: Schematic overview of a global-aware ICFG.

these initializations are not bundled in a function and do not represent instructions in LLVM IR. After considering the global variables, an artificial control-flow edge to the first global constructor C_0 must be introduced. The ICFG must determine the registered global constructors using the special `@llvm.global_ctors` variable, sort the functions according to their priority, analyze them, and introduce artificial control-flow edges between them. The return instructions of the first $n-1$ global constructors conceptually transfer control to the next global constructor. As the behavior is not defined whenever two global constructors have the same priority, it does not matter in which order the ICFG organizes them; user code is not allowed to depend on initialization order in that case. We denote global constructors as C_i in Figure 2. Global constructors and destructors can call arbitrary functions of the program, however, we do not represent that fact in Figure 2 to avoid cluttering. C^{UD} denotes the special function, introduced by the compiler, that calls the constructors and registers the respective destructors of global variables of *user-defined* types. Once all global constructors have been analyzed, the control flow is artificially transferred to the actual user-defined entry point(s), i.e., `main()` (and potentially a set of other entry points E_i). After constructing the control flows for the main program, an artificial control flow edge to the first global destructor D_0 are introduced. Similar to the constructors, the destructors are also chained according to their priority. The last global destructor transfers control flow to box that we denote as D^{ud} in Figure 2. D^{ud} is the sequence of calls to the destructors of global variables of user-defined types that have been registered in the C runtime. The D^{ud} destructors are called in reverse order of registration.

C. Data Flows

Similar to an ICFG algorithm that needs to be parameterized with a set of entry points, a data-flow solver needs to start at some program location(s). In the Soot [5] and PhASAR [2] frameworks, these program location(s) are referred to as *initial seeds*. Both frameworks allow analysis writers to specify the initial seeds by implementing a function of the appropriate problem interface that represents the analysis problem to be solved. The initial seeds mechanism allows analysis writers to not only specify the program locations but also data-flow facts that initially hold at these locations. The initial seeds implementation returns a mapping from start locations to a set of data-flow facts that hold initially.

Rather than using the flow-function implementations as described in Section III-A, the initial seeds can be used to model the effects of the primary initialization denoted by the I box in Figure 2. In addition to the ordinary initial seeds that an analysis writer specifies for their analysis, they can iterate the global variables and their primary initializations in order of occurrence and model the effects by creating a set of data-flow facts that represents the behavior according to their concrete analysis problem. The propagation of this set of data-flow facts that represents the global variables is then started at the beginning, i.e., the first statement, of C_0. Due to the artificial control-flow edges introduced in the global-aware ICFG, the flow facts are made available to the global constructors and the behavior of those constructors can be modeled soundly. After the solver propagated the flow facts through the code in the box labeled C, they now capture the effects of any initializing code and can then safely propagated into `main()` (and potential other user-defined initial seeds E_i). At the end of `main()`, the global variables are propagated through the chain of global destructors and destructors of global variables of user-defined types D^{ud} as indicated by the box labeled D.

To make the global variables available for analysis as data-flow facts at all statements, they are propagated into any potential call target at a given call site and propagated back to the caller at the callee's respective exit site(s). Data-flow facts that represent global variables must be killed at *call-to-return* flows to make the effects of callees visible to the subsequent program. In case only function declarations are available as call targets at a given call site, global variables are automatically killed by the *call* flow and instead must be propagated along the *call-to-return* flow. Otherwise, the globals would, again, not be available to the subsequent program. We discovered this special case while using the scheme presented here in a complex data-flow analysis we recently implemented.

IV. Implementation

We implemented the scheme presented in Section III within PhASAR [2]. We extended the existing LLVM-based ICFG implementation with functionalities that allow analysis writers to easily retrieve the global constructors and destructors. We also added an additional option to the ICFG's constructor to allow for ICFG construction that respects the functions that are called "before" *and* "after" the user-specified entry points, e.g. `main()`, and correctly reflects the actual semantics of global variables and their (de-)initializers. If enabled, the global initializers C_i and C^{ud} are analyzed first and registered destructors D^{ud} are recorded. The ICFG then adds artificial control-flow edges to the actual user-defined entry points. From the exit sites of the user-defined entry point functions artificial control-flow edges to the global destructors D_i and registered destructors D^{ud} are added as shown in Figure 2.

We also generalized the `initialSeeds()` implementation which, until now, has been shared across the IFDS [6] and

IDE [7] problem interfaces. This, however, prevented analysis writers from specifying data-flow facts with initial values other than $\bot$ in IDE problems making it impossible to encode the effects of primary global initializations within the `initialSeeds()` implementation. Our generalization now also allows for arbitrary initial edge functions in IDE [7].

PhASAR's pre-defined flow function implementations for automated parameter mapping for *call* and *return* flows have also been extended. We added additional parameters to the respective flow functions that allow for automatically handling the data flows of global variables as described in Section III-C.

V. Case Study: Constant Propagation

We demonstrate the usefulness of our PhASAR extension $\mathcal{G}^+$ by presenting how the new functionalities can be used to add global variable support to PhASAR's existing linear-constant propagation encoded within the IDE [7] framework. We then present a quantitative evaluation that assesses the importance of correctly handling global variables (and code).

A. An Analysis Writer's Perspective

When constructing the target program's ICFG to conduct a global-aware whole program analysis, we specify `main()` as an entry point and, in addition, turn on the option for global (de-)initializer awareness. The ICFG implementation then automatically analyzes the global code and introduces artificial control flows.

To capture the primary initialization (cf. I-labeled box Figure 2) we make use of the `initialSeeds()`. We iterate the global variables using LLVM's standard API and create G a set of pairs of variables and associated edge functions describing their initialization. The set, among others, includes $i \mapsto \lambda x.42, j \mapsto \lambda x.1024$. We query the ICFG for C_0 and return as initial seeds a mapping from C_0's first statement to G. We use the extended flow functions for automated handling of inter-procedural flows, i.e., *call*, *return*, *call-to-return*, and enable the option allowing for automated handling of global variables. The correct propagation is then automatically handled by PhASAR's solver implementation which propagates the data-flow facts according to the global-aware ICFG.

B. Global Variables in Real-World Programs

Our empirical evaluation addresses the research questions:

- **RQ1**: To what extend are global variables used in real-world programs?
- **RQ2**: How much precision does an analysis gain by making it global-aware?
- **RQ3**: What is the runtime cost of making an analysis global-aware?

To address **RQ1**, we counted the number of global variables for each benchmark program, recorded their respective types and determined their users by following their def-use chains. To address **RQ2**, we ran a global-oblivious as well as a global-aware IDE [7]-based linear-constant analysis that has been independently implemented in PhASAR on each benchmark target and compared the data-flow facts that have been generated and propagated by the analyses. We measured the analysis' running times to be able to comment on the expense that propagating the additional (global) variables incurs (**RQ3**).

TABLE I: Results for the IDE-based linear-constant analysis.

program	#g	#u	#gen	#ntvas		#ntvae		runtime [s]	
				$\mathcal{G}^+$	$\mathcal{G}^-$	$\mathcal{G}^+$	$\mathcal{G}^-$	$\mathcal{G}^+$	$\mathcal{G}^-$
bison	1,806	7,130	113	113	0	113	78	2582	2295
brotli	163	272	0	0	0	0	0	143	142
curl	1,880	2,119	17	17	0	17	8	730	698
file	168	267	5	5	0	4	0	1	1
gravity	1,194	3,333	17	17	0	16	10	60586	60482
grep	415	978	60	60	0	60	46	290	256
gzip	351	2,007	97	97	0	96	15	63	47
htop	1,521	2,355	44	44	0	41	20	632	596
libjpeg	184	346	0					19780	19989
libpng	454	560	0					97	114
libssh	1,853	1,997	7					1232	1301
libtiff	1,309	1,422	1					560	645
libvpx[d]	1,372	2,778	19	19	0	19	0	10645	10160
libvpx[e]	1,682	3,191	21	21	0	21	1	12558	11974
libxml2	4,969	8,475	92					28555	29689
libzmq	1,191	3,154	0					1866	2481
lrzip	782	1,415	4	4	0	4	4	250	252
lz4	396	1,189	13	13	0	13	5	115	108
openssl	1,835	1,899	14					1642	2005
openvpn	4,343	4,893	41	41	0	0	0	21979	21994
opus	467	606	2					415	516
tmux	5,193	5,916	40	40	0	0	0	22246	22333
xz	455	932	48	48	0	46	35	33	26

1) Experimental Setup: We have evaluated our framework extension $\mathcal{G}^+$ using as benchmark subjects 23 C/C++ programs that we obtained from Github. We compiled the programs to LLVM IR using WLLVM and subjected the resulting bitcode files to a linear-constant propagation, once using a global-oblivious $\mathcal{G}^-$ and once using a global-aware $\mathcal{G}^+$ version of the analysis. The target programs' corresponding LLVM IR ranges from 2,357 to 684,202 lines of code. We measured the running times for the experiments on an dual socket system with 2x Intel(R) Xeon(R) CPU E5-2630v4@2.20GHz CPUs and 256GB main memory, running Debian 10. We ran each experiment ten times and computed the mean time it took to execute the analysis. The mean relative standard deviation for all projects is 1.1%. Table I shows our results. The columns of the table present (from left to right) for each target subject the number of global variables, the number of their users, the number of global integer-typed variables that the analysis can potentially track, the number of constant variables that hold at the start of `main()` and the number of constant variables that hold at the end of `main()`—once using a globals-aware ($\mathcal{G}^+$) and once using the plain, unmodified constant analysis ($\mathcal{G}^-$)—as well as the running times in seconds. Our benchmark programs, the raw as well as the processed data produced in our evaluation are available in our artifact [4].

*2) **RQ1**: Usages of Global Variables:* Table I shows that all of our real-world target programs make use of global variables. The amount of global variables used ranges from 163 to 5,193 with an average of 1,478. These global variables, on average,

have 2,580 users. Global con- and destructors using the __ annotate__ keyword are used by two projects (libssh and libzmq) and thus seem to be used less frequently.

> Global variables are frequently used throughout all of our target subjects. Hence, it is important for an analysis to model them correctly.

*3) **RQ2**: Precision:* Our results for $\mathcal{G}^+$ show that most of the integer-typed global variables that are constant at the beginning of main() remain constant or linearly depending on constants throughout the whole main() function, i.e., the program. The openvpn and tmux programs present two exceptions where none of these variables remains constant. However, the results for $\mathcal{G}^-$ shows the necessity of handling global variables. Since global variable initialization is not taken into account by $\mathcal{G}^-$, it cannot find constant global variables at the beginning of main(). As the amount of constant globals at the end of main() indicates, there seem to be a few stores of constant values (or literals) to some of these globals. Still, the number of constant global variables at the end of the program lacks far behind $\mathcal{G}^+$.

> While ignoring global variables might be acceptable for analyses that are used for bug finding, especially analyses that are concerned with software security or are used as a basis for program optimizations cannot afford to ignore these variables (and respective code of global (de)initializers).

*4) **RQ3**: Performance:* As our results in Table I show, analyzing global variables impedes performances. This is because global variables need to be propagated through the complete program under analysis to represent the fact that they can be accessed (and modified) at any point in the program. Surprisingly, libraries benefit from our model. This is because PhASAR's points-to-based ICFG implementation and the global-oblivious analysis cause expensive repropagations when no dedicated main() function (or C and D control flows cf. Figure 2) is present and global variables are discovered. Besides the implementation effort, this behavior can be mitigated in which case we expect $\mathcal{G}^-$ to be slightly less expensive than $\mathcal{G}^+$ similar to the non-library target subjects.

> Supporting global variables impedes an analysis' performance. Making the IDE-based linear-constant analysis global-aware causes a performance hit of 7.5 % for ordinary programs and a performance gain of 12.6 % for libraries.

VI. Related Work

Modeling the effects of global variables in static analysis is a demanding task. Doing so in a sound manner is virtually impossible for many realistic target programs. However, current analysis frameworks such as Soot [5] and PhASAR [2] do not provide any framework support for modeling global variables.

Unfortunately, the compiler community does not provide solutions for comprehensive data-flow analysis of globals either. Optimizing compilers have to be rather conservative when it comes to performing code transformations, of course. While LLVM provides some optimizations w.r.t. global variables such as globalsmodref, constmerge, globalopt, and internalize, these are all rather simple analyses that back off as soon as a global variable's address is taken or its initialization is more complex. LLVM's implementations for (inter-procedural) constant propagation and constant folding does not optimize code that involves non-immutable globals with non-trivial (primary) initializers, and does not to aim to proof any properties for such variables or their users.

VII. Conclusion

In this paper, we have presented an overview on the complex semantics of global variables in C and C++ and how they map to LLVM's intermediate representation. Based on our observations, we presented a scheme that can be used to soundly model the effects of global variables in data-flow analysis. We extended the PhASAR [2] framework and implemented new functionalities that allow analysis writers to model globals in an easier and more structured manner. We presented a possible usage of the proposed scheme and showed its usefulness by extending PhASAR's current IDE-based linear-constant propagation adding support for global variables. Using the proposed scheme allows one to trivially add sound, full-global support to any data-flow analysis.

Acknowledgment

This work was partially supported by the Heinz Nixdorf Foundation and the German Research Foundation (DFG) within the Collaborative Research Centre 901 "On-The-Fly Computing" (grant no. 160364472-SFB901/3), project "Pervolution" (grant no. AP 206/11-1 and SI 2171/2-1), project "Green Configuration" (grant no. SI 2171/3-1), and project "Congruence" (grant no. AP 206/14-1).

References

[1] W. Wulf and M. Shaw, "Global variable considered harmful," *SIGPLAN Not.*, vol. 8, no. 2, p. 28–34, Feb. 1973. [Online]. Available: https://doi.org/10.1145/953353.953355

[2] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An inter-procedural static analysis framework for c/c++," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 393–410.

[3] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[4] Supplementary material. [Online]. Available: https://drive.google.com/drive/folders/1zYKsAXhg1Xt_qY-7GP9M7q3DLzk6PzEG?usp=sharing

[5] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, p. 13.

[6] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/199448.199462

[7] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Theor. Comput. Sci.*, vol. 167, no. 1-2, pp. 131–170, Oct. 1996. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(96)00072-2