

# Intraprocedural Analysis Based on Symbolic Execution for Bug Detection

A. E. Borodin<sup>a,\*</sup> and I. A. Dudina<sup>a,b,\*\*</sup>

<sup>a</sup> *Ivannikov Institute for System Programming, Russian Academy of Sciences,  
ul. Solzhenitsyna 25, Moscow, 109004 Russia*

<sup>b</sup> *Moscow State University, Moscow, 119991 Russia*

\**e-mail: alexey.borodin@ispras.ru*

\*\**e-mail: euparina@ispras.ru*

Received February 18, 2021; revised February 25, 2021; accepted March 12, 2021

**Abstract**—In this paper, we overview the approaches and techniques employed by the Svace static analysis tool for intraprocedural analysis. This analysis implies the traversal of the control flow graph, symbolic execution with state merging, analysis of a number of paths in functions with loops, simultaneous run of all analyzers, modeling of accessible memory cells, and value numbering.

DOI: 10.1134/S0361768821080028

## 1. INTRODUCTION

Svace is a static analysis tool for bug detection in C/C++/Java source code. Svace finds errors with a small number of false positives in a time comparable to compile time. The analysis does not require special preparation of programs and does not impose any constraints on language constructs used.

For bug detection, Svace uses different approaches: analysis based on an abstract syntax tree (AST) and interprocedural analysis with modeling of values of variables and memory cells.

As input, Svace receives the source code of the program together with a build command. The tool intercepts the compile and link commands. Then, a modified compiler (Clang for C/C++ [3] or OpenJDK javac for Java [4]) is run. The compiler constructs an AST to run bug detectors, as well as generates an intermediate representation of the program (LLVM bitcode [5] for C/C++ or bytecode for Java). The intermediate representation is fed to the SvEng<sup>1</sup> analyzer. The engine constructs a call graph and initiates a sequential analysis of each function, starting with the leaves of the graph. In this paper, we describe intraprocedural analysis of individual functions, which forms a basis of interprocedural analysis.

Section 2 describes the generalized analysis based on symbolic execution [6], which can be used by various analyzers to find errors in source codes of programs. The scheme of the analysis is outlined and its main abstractions (value identifier, reference, pointer

graph, attribute, and abstract state) are considered. Subsection 2.7 describes an extension of the analysis that enables path sensitivity. Section 3 describes its implementation in Svace for intraprocedural analysis of functions.

## 2. GENERALIZED ANALYZER

The developed procedure is intended primarily for non-sound analysis. By the non-sound analysis, we mean the analysis that can provide incorrect results in some cases to fulfill certain requirements (speed, memory consumption, ease of implementation, etc.).

### 2.1. Symbolic Execution with State Merging

Let us describe a bug detection analysis based on symbolic execution with union of states at path merge points.

Suppose that we have a control flow graph for some function. The edges of the graph represent abstract states, which describe certain properties of the function. Symbolic execution is carried out in topological order. For each vertex of the graph, from an abstract state on the incoming edge, the analysis forms an abstract state for its outgoing edge. Merge points of paths in acyclic subgraphs are analyzed once the states on the incoming edges are obtained. The analysis forms an abstract state on the outgoing edge of the function, which describes properties of the program for all paths under consideration.

To analyze strongly connected components (SCCs) in the control flow graph, several iterations of

<sup>1</sup> Svace Engine: the main engine of Svace.

the loop with preservation of all abstract states on the edges outgoing from SCCs are carried out. Once SCCs are analyzed, the states on the outgoing edges are merged. In this case, heuristics are employed to obtain a state that describes all possible execution paths of SCCs. However, if these heuristics work incorrectly, then the analysis can incorrectly describe properties of some paths.

If it is possible to extract an inner SCC from a SCC, then several iterations of the analysis are carried out for the inner SCC at each step of analyzing the outer SCC. This scheme of loop analysis has exponential complexity in the number of nested loops. To reduce the complexity for inner SCCs, the number of traversals is reduced. When a certain nesting threshold is reached, the analyzer stops to extract the inner SCCs.

We use the following notation:

- $S$  is a set of symbols;
- $I$  is a set of instructions;
- $P$  is a set of vertices in the control flow graph;
- $E$  is a set of edges in the control flow graph;
- $\Gamma$  is a set of abstract states.

The analysis is parameterized by the following components:

- additional analyses  $A_i$  on the basis of which warnings are issued;
- detectors  $C_i$  for finding program errors;
- transfer functions for each instruction  $T[I]$ ;
- number of SCC traversals  $N$ ;
- functions that create a state at path merge points  $U[P] : \Gamma \times \Gamma \mapsto \Gamma$ ;
- functions that create a state from several states for outgoing edges of SCCs  $U[P] : \Gamma \times \Gamma \mapsto \Gamma$ .

For each instruction, all additional analyses and detectors are run simultaneously.

The detector is run for each traversed vertex and, based on the abstract state on its incoming edge, issues an error warning. The additional analyses form abstract states on the outgoing edges.

When traversing a vertex, any implemented analyzer or detector has access to the results of all other analyzers on the incoming edges of this vertex, but it does not have access to the results on its outgoing edges. It is important that the sequence of analyses does not affect the result.

This implementation has the following advantages.

- High speed of analysis; general actions are performed by one analyzer.
- Rather complex detectors can easily be implemented because all analyzed properties are available to each detector.

## 2.2. Value Identifiers

A *value identifier*<sup>2</sup> is an abstraction that is used to group values of variables into equivalence classes for a symbolic execution step and obeys the following rules.

I. If two variables are assigned the same value identifier, then these variables have the same values at runtime (the value numbering problem<sup>3</sup>).

II. For an instruction, the value identifiers on the incoming and outgoing edges are identical for all variables and memory cells, except for those the values of which vary in this instruction. This requirement for value identifiers significantly facilitates the implementation of transfer functions: all created properties do not lose their correctness in the process of analysis.

Value identifiers play a key role in the proposed analysis. In addition to value numbering, value identifiers are used to describe most of the analyzed properties of values. In this case, the properties are associated with value identifiers. In turn, the properties can be described using value identifiers.

To describe properties, *attributes* are used. Attributes can be associated with value identifiers and with edges of the control flow graph for certain abstract states. An attribute represents an analyzed property (value interval of a variable, necessary reachability condition, list of value identifiers for locked mutexes, etc.). Attributes must have function  $\sqcup$  that merges two attributes. This function is used for state merging. In many cases, it is convenient to represent attributes as lattices (or semi-lattices) and represent the merge function as the least upper bound. However, this is not a requirement for attributes.

Attributes allow an abstract state to be shared among additional analyzers: each analyzer has its own set of attributes, which is why there are no conflicts when several analyzers form an output state in different ways.

An abstract state contains information about values of all attributes for value identifiers. For illustrative purposes, let us consider the following attributes: value interval  $VI$ , which describes the range of possible values for a variable, and attribute *Null*, which indicates that a variable has a zero value. The type of an attribute is written in bold, while the interval value is italicized:  $VI$  is the type of an attribute and  $VI$  is the set of values.

By  $V$ , we denote a set of value identifiers. Function  $val: S \hookrightarrow V$  returns an associated value identifier for each symbol. Function  $val$  is a part of an abstract state. An abstract state also contains the value of an attribute for each attribute type and value identifier:

<sup>2</sup> The value identifier is a symbolic variable. In this paper, we use this term to denote symbolic variables together with a way of their use in the analyzer.

<sup>3</sup> The value numbering problem consists in determining a set of variables the values of which are equal at a certain point for all execution paths. In the case of symbolic execution, all execution paths can be replaced by all paths under consideration.

- $\Gamma[VI]:V \mapsto VI$ ,
- $\Gamma[Null]:V \mapsto Null$ .

Instead of  $\Gamma(Null, V)$ , we write  $\Gamma[Null](V)$  or  $Null(V)$ .

Let us consider the following code fragment in C that illustrates the benefits of associating attributes with value identifiers rather than with program variables.

```
1: void func(int*p) {
2:   int*q = p;
3:   if(!q) {
4:     *p = 7;
5:   }
6: }
```

Listing 1. Null pointer dereference.

Upon analyzing the instruction in line 2,  $val(q) = val(p)$  is executed in an abstract state. Suppose that  $val(q) = v_1$ . When analyzing the instruction in line 3, the value identifier of variable  $q$  is associated with attribute value  $Null$ , which indicates that a pointer has only a null value. When analyzing the dereference statement in line 4, the abstract state has the following properties:

$$val(p) = v_1, \quad val(q) = v_1, \quad \Gamma[Null](v_1) = null.$$

In fact, the analyzer knows that the pointer that can have only a null value is dereferenced. This is sufficient to issue a null pointer dereference warning. If line 4 is reachable, then an error occurs.

With the value identifiers representing the values invariant at different points of the program, many properties can be expressed in terms of value identifiers. In other words, attribute values refer to value identifiers. In this case, value identifiers are used as symbols of a certain alphabet.

Let us consider attribute  $pt$  (see Subsection 2.4) the values of which are value identifiers for addresses of pointed cells.

```
1: void func(int f) {
2:   int a, b, c;
3:   int*p = f>0? &a : f<0? &b : &c;
4:   int*q = p;
5:   p = 0;
6: }
```

Listing 2. Example with assignment.

The state at point 3 in Listing 2 has the following properties:

$$val(\&a) = v_a, \quad val(\&b) = v_b, \quad val(\&c) = v_c,$$

$$val(p) = v_{pp}, \quad pt(v_{pp}) = \{v_a, v_b, v_c\}.$$

Thus, the value of variable  $p$  refers to the addresses of variables  $a$ ,  $b$ , and  $c$ .

The same properties also hold for line 6, even though the value of variable  $p$  has changed:

$$val(p) = v_0, \quad val(q) = v_{pp}, \quad pt(v_{pp}) = \{v_a, v_b, v_c\},$$

$$\Gamma[Null](v_0) = null.$$

Thus, it is sufficient for the analyzer to change the information about the value of  $p$  without modifying the properties of its values, which makes it possible to optimize the time of the analysis.

### 2.3. References

By *references*, we mean a subset of value identifiers for which the following two conditions hold:

(i) the value is a pointer;

(ii) the analyzer performs memory modeling for this pointer, i.e., the values in the pointed memory cell are tracked.

Let us extend function  $val$  for the purpose of memory modeling:  $val: S \cup R \hookrightarrow V$ , where  $R$  is a set of references. Notation  $val(s) \mapsto v_1$ ,  $val(v_1) \mapsto v_2$  means that variable  $s$  has a value described by identifier  $v_1$  and the value in the cell to which  $s$  points is described by identifier  $v_2$ .

References are used to model pointers, and they always correspond to pointers. Only the pointers for which the analyzer can determine that they are not aliases are modeled. In the simplest implementation, references denote addresses of local variables that are a priori not equal.

### 2.4. Pointer Graph

Analysis of pointers allows one to determine to which modeled memory cells variables point. The result of this analysis can be represented as a pointer graph.

The pointer graph is a directed graph  $P = \langle V, pt \rangle$  the vertices of which are value identifiers and the edges of which go from value identifiers to references. The edges of the graph are defined by function  $pt: V \hookrightarrow 2^R$ , which returns a set of pointed references for a value identifier. If the pointer graph has edge  $\langle v_1, r_2 \rangle$ , then this means that the value represented by identifier  $v_1$  points to a memory cell modeled by reference  $r_2$ . In other words, value  $v_1$  can have alias  $r_2$ .

Function  $pt$  is another part of an abstract state. Analysis of pointers is required to model indirect memory access. There are no specific requirements for this analysis, and it can be regarded as another parameterization of the proposed analysis procedure.

### 2.5. Strong and Weak Updates

If the analysis of an instruction that assigns a value to a variable or memory cell can eliminate the effect from the assignment of all previous values, then this behavior is called a strong update. If the effects of the

previous assignments are still taken into account by the analyzer, then it is called a weak update.

In the proposed analysis, strong updates can be carried out for all variables and cells for which the pointer graph contains only one outgoing edge.

Let us consider the following C instructions for indirect memory access:  $r = *p$  and  $*p = a$ . We describe transfer functions for these instructions in the case of an arbitrary analysis that stores its results in attribute  $A_i$ .

The memory read instruction is  $m_n = *p$ . Suppose that  $pt(val(p)) = Pt$ . There are three possible cases: set  $Pt$  has one element, the set has more than one element, and the set is empty. In the first case,  $Pt = \{m\}$ , a strong update can be carried out; as a result, the output state contains  $val(r) = val(m)$ .

In the case of more than one element,  $Pt = \{m_1, m_2, \dots, m_n\}$ , a new value identifier  $v_r$  is created and associated with  $r$  in the output state,  $val(r) = val(v_r)$ . For all attributes, we use the following property merge (join) function:

$$\Gamma[A_i](v_r) = \sqcup (\Gamma[A_i]val(m_1), \dots, \Gamma[A_i]val(m_n)).$$

If the set is empty, then a new value identifier is associated with variable  $r$ .

The memory write instruction is  $*p = a$ . Let us also consider three cases for set  $Pt$ . If  $Pt = \{m\}$ , then  $val(m) = val(a)$  in the output state. In the case of  $Pt = \{m_1, m_2, \dots, m_n\}$ , a weak update is carried out. For this purpose, a new value identifier  $v_i$  is created for each memory cell; this identifier is associated with the cells in the output state,  $val(m_i) = v_i$ . For all attributes, we use the following property merge (join) function:

$$\Gamma[A_i](v_r) = \Gamma[A_i](val(a) \sqcup \Gamma[A_i](prev(m_i))).$$

Here, function *prev* returns value  $val(m_i)$  (if it is defined) or creates a new value identifier. Thus, for each possible reference, the fact that its value could have changed is taken into account. If the set is empty, then a new value identifier is associated with variable  $val(p)$ .

To create an abstract state at a path merge point, a rule similar to that for the memory read instruction is used. The rule for values of modeled memory cells merges properties from input states

$$\begin{aligned} & \Gamma[A_i](journal(m)) \\ &= \Gamma_1[A_i](val(m) \sqcup \Gamma_2[A_i](prev(m_i))). \end{aligned}$$

Function *journal* returns a value identifier for reference  $m$  if, in both states for the incoming edges, the same value identifier is assigned to  $m$ ; otherwise, it creates a new value identifier.

## 2.6. Analysis Core and Additional Analyses

All additional analyses are implemented as instruction handlers that operate with value identifiers rather than with variables. The main engine tracks the pointer graph, executes strong or weak updates, and calls handlers for the corresponding situations. In fact, the main engine processes the following components of abstract states: *val*, *pt*; i.e., it tracks values of variables and performs analysis of pointers.

Let us consider a small example in Listing 3, where interval analysis is used as additional analyses.

```
1: int f(int a, int *p) {
2:   int x = 1;
3:   *p = x;
4:   if (a)
5:     *p = 2;
6:   return *p; }
```

Listing 3. Small example.

The abstract states of the analysis after the corresponding lines are

$$2: val(x) = v_x, VI(v_x) = [1, 1];$$

$$3: \Gamma_2, val(p) = v_p, val(v_p) = v_x;$$

$$5: \Gamma_4, val(v_p) = v_2, VI(v_2) = [1, 2];$$

$$6: \Gamma_5, val(v_p) = v_{p2}, val(ret) = v_{p2}, VI(v_{p2}) = [1, 2].$$

The value of attribute *VI* in the last abstract state is obtained by applying the attribute merge (join) function in accordance with the rule described in Subsection 2.5 for merge points:

$$\begin{aligned} & \Gamma[VI](journal(v_{p2})) \\ &= \Gamma_1[VI](val(v_p) \sqcup \Gamma_2[VI](prev(v_p))), \end{aligned}$$

$$\begin{aligned} & \Gamma[A_i](journal(v_{p2})) \\ &= \Gamma_1[A_i](val(v_x) \sqcup \Gamma_2[A_i](prev(v_2))) \\ &= [1; 1] \sqcup [2; 2] = [1; 2]. \end{aligned}$$

## 2.7. Path Sensitivity

An analysis is called path-sensitive if it is capable of distinguishing execution paths of a program. The implementation of path sensitivity described below is based on expanding the use of value identifiers, which are regarded as building blocks in conditional expressions and in definitions of value identifiers. The goal of path sensitivity is to filter out the paths not feasible due to inconsistent conditions.

For a particular analyzer, adding path sensitivity not only reduces the number of false warnings due to inconsistent conditions but also increases the number of true warnings. The latter is achieved by issuing warnings in complex cases where an analyzer without path sensitivity cannot provide an acceptable quality of analysis.

For instance, in Listing 4, the path that passes through two dereferences of  $p$  is not feasible.

```

1: void f(int a, int**p) {
2:   int x = a+2;
4:   if(a>10) {
5:     *p = 0;
6:   }
7:   if(x<12) {
8:     **p = 0;
9:   }
10: }
```

Listing 4. Function with an infeasible path.

If the path were feasible, then a null pointer  $p$  dereference would occur in line 8.

An attribute that characterizes conditions of occurrence of certain events is called a conditional attribute. The value of an attribute is a formula, an expression of propositional logic, where programming language constants and value identifiers can be used to express properties of values. An example of a condition is  $(v_x > v_y) \wedge (v_x \neq 0) \vee (v_y < 10)$ .

There is a predefined conditional attribute *Ness*: the necessary conditions for the reachability of an edge in the control flow graph.

The other attributes are used by individual detectors in accordance with the following scheme. At the point where an event of interest occurs, the value of attribute  $C_i$  is set to *true* and is associated with a certain value identifier. At path merge points 1 and 2, the value of the attribute is computed using the following formula:

$$\Gamma_{res}[C_i](v) = (\Gamma_1[Ness](v) \wedge \Gamma_1[C_i](v)) \vee (\Gamma_2[Ness](v) \wedge \Gamma_2[C_i](v)).$$

To filter out infeasible paths, an SMT solver is run before issuing an error warning; as input, the solver receives a conjunction between the necessary condition and the condition of a tracked attribute. If the SMT solver returns *unsat*, then there is no error on these paths.

For the example from Listing 4, the value of attribute *Ness* is  $v_x = v_a + 2 \wedge v_x < 12$ , and the value of the attribute that tracks null pointer assignment is  $v_a > 10$ . With formula  $v_x = v_a + 2 \wedge v_x < 12 \wedge v_a > 10$  having no model, the SMT solver ensures that no false warning about null dereference is issued.

It should be noted that transfer functions for analyzed properties and necessary conditions, as well as the form of conditional attributes, depend on their particular implementation. A simpler implementation may not add condition  $v_x = v_a + 2 \wedge v_x < 12$  to the set of required conditions. In this case, the path is not filtered out. In fact, this is a path-sensitive analysis without data sensitivity.

The SMT solver is called only when an error is suspected. If the formula is not decidable, then the warning is not issued; in all other cases, the warning is issued. The SMT solver is not called to compute intermediate data.

The implementation of the path-sensitive analysis described above has the following advantages:

- ease of extension for the analysis based on value identifiers;
- high speed, because the SMT solver is called only when an error is suspected;
- no constraints on types of analyzed formulas are imposed.

### 3. IMPLEMENTATION OF THE ANALYSIS IN SVACE

#### 3.1. Heuristics

The analysis is non-sound and can use heuristics both to improve its accuracy (sometimes to the detriment of correctness) and to reduce the analysis time. The main heuristics are as follows:

- the input parameters of the procedure, including their offsets and dereferences, are not aliases;
- a selected set of paths of the analysis describes all its essential paths (a path is considered essential if it can affect the result of the analysis).

The analysis covers all possible paths in functions without loops and only some paths in functions with loops.

In addition, the following constraints on various parameters are imposed:

- the body of a loop is traversed only twice (a larger number of traversals slows down the analysis);
- the maximum number of references modeled for one value identifier is 150;
- the maximum number of non-constant offsets modeled for one pointer is 10;
- the maximum length of a chain of dereferences and offsets modeled for variables is 6;
- the maximum analysis time for one function is 5 min.

The constraint on the analysis time for one function is used to protect against non-standard code. For all well-known projects, the analysis of all functions fits into the 5-min time limit.

#### 3.2. Analyzed Language: *svace0*

The analysis is carried out for the internal representation in the *svace0* language, which is a simplified version of the LLVM language with additional instructions that make it possible to gather more information about a program.

**Table 1.** Analysis time for big projects

Project	Code size, thousand lines	Analysis time, min		Build time, min
		server1	server2	
tizen 5.5	19988	272	516	250
android 5	8561	236	421	31
android 9 java	12122	27	32	77

To analyze programs in C/C++, they are translated into the intermediate representation of LLVM, which is then translated into svace0.

To analyze programs in Java, they are translated into bytecode, which is then translated into svace0. The implementation of the majority of detectors and analyzers for C/C++ and Java is similar.

This is a quite low-level language. On the one hand, this makes it possible to adequately model the semantics of programs; on the other hand, this complicates the analysis of some high-level constructs.

### 3.3. Attributes

The simultaneous run of all analyzers and the possibility of using the results of other analyzers provide a gain in both analysis time and memory consumption. The high speed of the analysis is due to the absence of redundant computations: the results of other analyzers can be used immediately. Memory optimization is achieved by the fact that the analysis does not need to store the majority of abstract states: in the general case, once an instruction is analyzed, the state on the incoming edge is no longer required and can be deleted.

The results of the analysis are saved as attributes. Adding a new attribute does not require significant computational effort. Currently, there are more than 350 attributes implemented in Svace. Below are some of their types:

- possible value interval for integer variables;
- array size interval and offset interval for array pointers;
- mutex is locked;
- variable is received from an unverified source;
- line length interval;
- heap pointer is compared with a constant;
- heap allocation conditions;
- conditions under which a variable is not initialized;
- conditions under which a pointer is assigned a null value (required to find null pointer dereferences);
- conditions under which a variable can have a zero value (required to find divide-by-zero errors);
- necessary reachability conditions for a point in a program.

### 3.4. Other Analyses

In Svace, the analysis procedure described above is only a small component used for intraprocedural analysis, which, in turn, forms a basis for summary-based interprocedural analysis. Based on interprocedural analysis, the analysis of constructors, destructors, and class assignment operators for C++ is carried out to detect inconsistencies in their implementation [7].

In addition, before initiating the intraprocedural analysis, a data flow analysis is carried out for each function; it computes important properties of the function (unreachable code, termination functions, live variables, etc.) in a conservative way [8].

Moreover, AST-based analyses are used to implement some of the detectors. These analyses are run in modified compilers (clang and javac).

### 3.5. Results

Table 1 shows the analysis times for some open source projects. Only the runtime of SvEng was measured. In the process of analysis, all implemented detectors were enabled.

The analysis was carried out on two servers with the following characteristics:

- server 1: Intel Xeon E5-2650 2.00 GHz, 32 cores, 256 GB RAM;
- server 2: Intel Core i7-6700 3.40 GHz, 8 cores, 32 GB RAM.

The source code size of the tizen and android operating systems is significant, which is why it is reasonable to analyze them on a server with at least 32 GB of RAM. The current speed of the analysis allows these projects to be analyzed during a nightly build. For comparison, Table 1 shows the build times for these projects on server 1.

For many small projects, the analysis requires not more than 2 GB RAM. Table 2 shows the data for the busybox, cairo, xorg-server, and nss projects (which amount to 139 to 393 thousand lines of code) on server 1 with limited memory.

The size of source code (in lines) allows us to roughly estimate the complexity of a project. The speed of the analysis depends on the complexity of code and constructs used. It can be seen that the speed of the analysis varies from 448 to 1534 lines per second with 16 GB RAM. Reducing the amount of RAM to

**Table 2.** Analysis time for small projects

Memory, GB	Project			
	busybox-1.18.5	cairo-1.12.14	xorg-server-1.12.3	nss-3.17.4
	Code size, thousand lines			
	139	270	393	355
16	310	176	524	336
8	314	160	521	336
4	319	156	520	335
2	338	158	594	378
1	419	231	1430	418

2 GB has only a slight effect on the analysis time. The worst result (274 lines per second) was achieved for xorg-server with 1 GB RAM; increasing the amount of RAM to 2 GB speeds up the analysis of this project by a factor of 2.4.

An important characteristic of the analyzer is the percentage of true warnings. In fact, the goal is to issue as many warnings as possible while ensuring an acceptable level of true positives. Table 3 describes the quality of warnings for 190 stable detectors. The table includes only the data for SvEng. The “Total” column shows the total number of warnings issued for the evaluated detectors, the “Labeled” column contains the percentage of manually labeled warnings among the issued warnings, and the “True” column provides the percentage of true warnings among the labeled ones.

#### 4. RELATED WORKS

Tools based on symbolic execution without state merging are PREFIX [9], Archer [10], and, apparently, Prevent [11]. PREFIX, which analyzes 50 paths by default, was the earliest implementation. The number of issued warnings almost ceases to vary upon considering more than 100 paths in a function.

The use of symbolic execution without path merging has both pros and cons. An obvious disadvantage is the problem of exponential growth of paths in the function. Because of this, the tool fails to analyze a sig-

nificant portion of paths in functions with a large number of paths. In addition, the total time of the analysis is higher as compared to the path merging scheme. Another problem is creating a summary for a function under analysis. For this purpose, it is required either to implement an additional analysis or confine oneself to a summary that incompletely describes the behavior of the function (because not all paths are analyzed). The advantages of this approach are as follows: each path is modeled more accurately, there is no need for a heuristic function to merge abstract states, and the implementation of many detectors is facilitated. In addition, it is easier to report an error to the user because it is sufficient to show the path under analysis.

The Saturn tool [12] unites states at path merge points. A SAT solver is used. Detectors are run successively. A loop analysis method depends (among other things) on the detector. Abstract states are not shared among detectors. Duration of analysis depends significantly on the enabled detectors. The Calysto tool [13], which is based on the SMT solver, has a similar architecture but higher speed and accuracy.

SharpChecker [14, 15] is a tool developed at the Ivannikov Institute for System Programming of the Russian Academy of Sciences to analyze C# programs. The tool is quite similar to the scheme described above (state merging, simultaneous run of all analyzers, and modeling of accessible memory cells), even though it was originally designed to be used in combination with an SMT solver. The current loop analysis scheme implemented in Svace was borrowed from this work. In the earlier versions of Svace, states were merged on back edges.

#### 5. CONCLUSIONS

In this paper, we have considered the intraprocedural analysis that enables a relatively fast and high-quality processing of large amounts of source code, which is confirmed by its implementation in Svace.

The general scheme of the analysis has been described. It can be improved in the following ways:

**Table 3.** Quality of analysis

Project	Total	Labeled	True
cairo-1.12.14	321	10.9%	94.2%
xorg-server-1.12.3	791	10.1%	78.7%
nss-3.17.4	826	22%	85.1%
busybox-1.18.5	1561	10.1%	80.5%
android 9 java	7327	10.68%	77.7%
android 5.02	10414	10.7%	76.4%
tizen 5.5	18920	22.1%	70.8%

- adding devirtualization to construct a more accurate call graph;
- improving the analysis of pointers, including the use of path-sensitive pointer analysis;
- more accurate modeling of loops;
- reducing the number of modeled memory cells and values of variables to speed up the analysis.

## REFERENCES

1. Ivannikov, V.P., Belevantsev, A.A., Borodin, A.E., Ignatyev, V.N., Zhurikhin, D.M., Avetisyan, A.I., and Leonov, M.I., Static analyzer Svace for finding of defects in program source code, *Tr. Inst. Sist. Program. Ross. Akad. Nauk* (Proc. Inst. Syst. Program. Russ. Acad. Sci.), 2014, vol. 26, no. 1, pp. 231–250. [https://doi.org/10.15514/ISPRAS-2014-26\(1\)-7](https://doi.org/10.15514/ISPRAS-2014-26(1)-7)
2. Borodin, A., Belevantsev, A., Zhurikhin, D., and Izbyshchikov, A., Deterministic static analysis, *Proc. Ivannikov Memorial Workshop*, 2018, pp. 10–14. <https://doi.org/10.1109/IVMEM.2018.00009>
3. Clang project. <https://clang.llvm.org>. Accessed September 10, 2020.
4. The javac compiler. <https://docs.oracle.com/en/java/javase/11/tools/javac.html>. Accessed September 10, 2020.
5. LLVM bitcode. <https://releases.llvm.org/8.0.1/docs/BitCodeFormat.html>. Accessed September 10, 2020.
6. King, J.C., Symbolic execution and program testing, *Commun. ACM*, 1976, vol. 19, no. 7, pp. 385–394.
7. Borodin, A.E. and Belevantsev, A.A., A static analysis tool Svace as a collection of analyzers with various complexity levels, *Tr. Inst. Sist. Program. Ross. Akad. Nauk* (Proc. Inst. Syst. Program. Russ. Acad. Sci.), 2015, vol. 27, no. 6, pp. 111–134. [https://doi.org/10.15514/ISPRAS-2015-27\(6\)-8](https://doi.org/10.15514/ISPRAS-2015-27(6)-8)
8. Mulyukov, R.R. and Borodin, A.E., Using unreachable code analysis in static analysis tool for finding defects in source code, *Tr. Inst. Sist. Program. Ross. Akad. Nauk* (Proc. Inst. Syst. Program. Russ. Acad. Sci.), 2016, vol. 28, no. 5, pp. 145–158. [https://doi.org/10.15514/ISPRAS-2016-28\(5\)-9](https://doi.org/10.15514/ISPRAS-2016-28(5)-9)
9. Bush, W.R., Pincus, J.D., and Sielaff, D.J., A static analyzer for finding dynamic programming errors, *Software-Pract. Exper.*, 2000, vol. 30, no. 7, pp. 775–802.
10. Xie, Y., Chou, A., and Engler, D., Archer: Using symbolic, path-sensitive analysis to detect memory access errors, *ACM SIGSOFT Software Eng. Notes*, 2003, vol. 28, no. 5, pp. 327–336.
11. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D., A few billion lines of code later: Using static analysis to find bugs in the real world, *Commun. ACM*, 2010, vol. 53, no. 2, pp. 66–75.
12. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., and Hawkins, P., An overview of the saturn project, *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Engineering*, 2007, pp. 43–48.
13. Babic, D. and Hu, A.J., Calysto: Scalable and precise extended static checking, *Proc. 30th Int. Conf. Software Engineering*, 2008, pp. 211–220.
14. Koshelev, V.K., Dudina, I.A., Ignatyev, V.N., and Borzilov, A.I., Path-sensitive bug detection analysis of C# program illustrated by null pointer dereference, *Tr. Inst. Sist. Program. Ross. Akad. Nauk* (Proc. Inst. Syst. Program. Russ. Acad. Sci.), 2015, vol. 27, no. 5, pp. 59–86. [https://doi.org/10.15514/ISPRAS-2015-27\(5\)-5](https://doi.org/10.15514/ISPRAS-2015-27(5)-5)
15. Koshelev, V., Ignatiev, V., Borzilov, A., and Belevantsev, A., Sharpchecker: Static analysis tool for C# programs, *Program. Comput. Software*, 2017, vol. 43, no. 4, pp. 268–276. <https://doi.org/10.1134/S0361768817040041>

*Translated by Yu. Kornienko*