ISSN 0361-7688, Programming and Computer Software, 2021, Vol. 47, No. 6, pp. 466–481. © Pleiades Publishing, Ltd., 2021.
Russian Text © The Author(s), 2021, published in Programmirovanie, 2021, Vol. 47, No. 6.

# Searching for Taint Vulnerabilities with Svace Static Analysis Tool

**A. E. Borodin[a,*], A. V. Goremykin[a,b,**], S. P. Vartanov[a,***], and A. A. Belevantsev[a,b,****]**

[a] *Ivannikov Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia*

[b] *Moscow State University, Moscow, 119991 Russia*

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

***e-mail: alexey.goremykin@ispras.ru*

****e-mail: svartanov@ispras.ru*

*****e-mail: abel@ispras.ru*

Received July 5, 2021; revised July 16, 2021; accepted July 22, 2021

**Abstract**—This paper is dedicated to finding taint-based errors in the source code of programs, i.e., errors caused by unsafe use of data from external sources, which could potentially be modified by a malefactor. The interprocedural static analyzer Svace is used as a basis. The analyzer searches for both program defects and suspicious points where the logic of the program may be corrupted. The goal is to find as many errors as possible at an acceptable speed and low false positive rate (<20–35%). For this purpose, Svace builds, with the help of a modified compiler, a low-level typed intermediate representation, which is input to the main SvEng analyzer. The analyzer constructs a call graph and then carries out summary-based analysis. In this analysis, functions are traversed according to the call graph, starting from the leaves. Once a function is analyzed, its summary is created, which is then used to analyze call instructions. The analysis has both high speed and good scalability. Intraprocedural analysis is based on symbolic execution with state merging at join points. An SMT solver can be used to filter out infeasible paths for some checkers. In this case, the SMT solver is called only if an error is suspected. The analyzer has been extended to find defects of tainted data use. The checkers are implemented as plugins based on the source–sink scheme. The sources are calls of library functions that receive data from the outside of the program, as well as arguments of the main function. The sinks are accesses to arrays, uses of variables as steps or loop boundaries, and calls of functions that require checked arguments. Checkers that cover most of possible vulnerabilities for tainted integers and strings are implemented. To assess the coverage, the Juliet project is used. The false negative rate ranges from 46.31% to 81.17% with a small number of false positives.

**DOI:** 10.1134/S0361768821060037

## 1. INTRODUCTION

This paper describes an implementation of a procedure for finding taint-based errors based on the Svace static analyzer [1–4].

The most important features of Svace are

• summary-based interprocedural analysis, whereby functions are traversed according to the call graph, starting from the leaves (each function is analyzed only once);

• non-sound analysis: by abandoning soundness, the analyzer improves its accuracy and performance;

• analysis in one function is based on value analysis: the analyzer tracks values of variables and memory cells and associates the majority of properties with values of variables.

Section 2 describes types of errors found by the analyzer. Sections 3 and 4 consider the design of the Svace static analyzer and the SvEng module, respectively. Section 5 discusses the implementation of the taint analysis, while Section 6 assesses the results on the Juliet and Tizen 6 projects.

## 2. TAINTED DATA

In this paper, we consider errors caused by unsafe use of data from external sources, which could potentially be modified by a malefactor. If these data are used without proper check, then the program has vulnerability.

Tainted data are data from files, user input, and data transmitted over the network. We consider

tainted data of two types: tainted integers and tainted strings.

Below are the types of vulnerabilities that are due to tainted data use.

• When accessing an array, a buffer overflow occurs, which could allow a malefactor to seize control of a device [5]. According to the U.S. National Vulnerability Database (NVD), errors of this type cause 9.49% of all vulnerabilities listed in the CWE database in 2018 [6].

• If tainted data are used as a loop constraint, then the loop may be executed more times than expected. This can cause a waste of CPU time and other errors, e.g., buffer overflow. If tainted data are used as a step of a loop iterator, then the data can be selected in such a way that the loop becomes infinite.

• When tainted data are used for some operations, e.g., memory allocation, a malefactor can force a program to allocate an excessive amount of memory.

Listing 1 illustrates vulnerabilities due to tainted integers. To fix the buffer overflow vulnerability, it is required to check whether the range of variable *n* falls within interval [0; 99]. Safe ranges for vulnerabilities of other types depend on the logic of a program.

Tainted strings can contain arbitrary characters or have arbitrary length. When copying this string to a fixed-size array, an array overflow can occur.

```
char buf[100];

int n;
scanf("%d", &n); //n is tainted

// buffer overflow occurs if n is less
than zero or greater than 99
buf[n] = 0;

//memory allocation
char*p = (char*)malloc(n * sizeof(struct
Fmt));
int i = 99;

while(i > 0) {
 buf[i] = '0' + (i % 10);
 //infinite loop occurs if n is 0
 i -= n;
}
```

**Listing 1.** Tainted integers.

The code in Listing 2 illustrates vulnerabilities due to tainted strings.

## 3. SVACE STATIC ANALYZER

Svace finds defects in source codes, including errors that can occur at runtime and suspicious points where program logic may be corrupted. The goal is to find as many defects as possible at an acceptable speed and low false positive rate.

The tool may miss real defects or yield false positive results, which do not correspond to real defects. For a program under analysis, no preparation is required: it is sufficient that its source code be compilable. The analysis time is comparable to the compile time of the program. For large programs, it is reasonable to run Svace during a nightly build.

The defect detection problem is undecidable [7]; i.e., it is impossible to find all errors of a certain type in an arbitrary program without false positives. This problem is solved by making various compromises. One of the approaches is to find an approximate solution. There are two types of approximations.

• Approximation towards the absence of false positives. In this case, only true positive warnings are issued; however, many defects can be missed. A typical example is compiler errors: it is unacceptable for the compiler to refuse processing a correct program.

• Search for all errors of a certain type. In this case, the false positive rate can be high. The quality of analysis can be improved by significantly increasing its runtime.

These approximations are called sound ones because they always round off the solution to one side. Svace does not use sound approximations, which allows it both to speed up the analysis of programs and to significantly improve the true positive rate.

In [8], it was stated that the sound analysis of aliases is necessary for program optimization while being optional for analyzers. When the authors of the thread- and context-sensitive analysis [9, 10] removed one step responsible for soundness, the overall time of the analysis was significantly reduced (in one case, from several days to several minutes).

Figure 1 illustrates possible approaches. The *Y*-axis represents the percentage of detected errors, while the X-axis represents the percentage of true positives. The approach implemented in Svace maximizes the area of the dotted rectangle.

This approach is quite popular and is not Svace's know-how. In [11], the term "soundy" was proposed to define a generally sound analysis that abandons soundness for some constructs.

```
char*p = getenv("aaa");

//potential buffer overflow
//size p may be less than
10
char x = p[10];

char buf[10];
int n = *((int*)p);
//buffer overflow
buf[n] = 0;
```

**Listing 2.** Tainted strings.

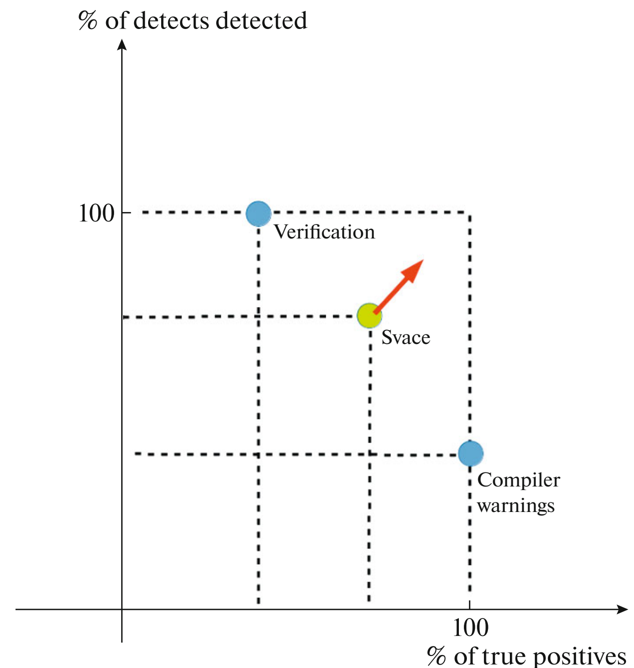


**Fig. 1.** Approaches to defect detection.

### *3.1. Architecture of Svace*

To analyze a program, the analyzer needs its source code and a build script. Svace supports defect detection for C, C++, Java, Kotlin, and Go.[1]

A typical analysis scheme is shown in Fig. 2. Svace intercepts the compile and link commands [12]. Then, a modified compiler is run to construct an abstract syntax tree (AST), checkers are launched to find errors on the AST, and an intermediate representation of the program is generated for further analysis. The intermediate representation is input to the SvEng analyzer,[2] which carries out interprocedural analysis.

AST-based analyzers traverse the AST nodes and carry out relatively simple rule checks. The AST analyzers can be used to find suspicious patterns in parse trees and various typos; at the same time,

• it is difficult to trace dependences between variables,

• it is difficult to analyze pointers, and

• it is difficult to carry out interprocedural and intermodular analysis.

Since searching for taint-based errors generally requires analyzing the properties mentioned above, we do not implement taint checkers at this level.

[1] Svace with Go support was released in January 2021.
[2] Abbreviated form of **Sv**ace **Eng**ine.

## 4. SvEng ANALYZER

### *4.1. General Scheme*

SvEng carries out deep thread- and context-sensitive interprocedural analysis. The most important features of SvEng are as follows.

• Summary-based interprocedural analysis, whereby functions are traversed according to the call graph, starting from the leaves. Each function is analyzed only once. Once a function is analyzed, its summary is created, which is then used to analyze call instructions. The summary describes the properties of the function. The analysis preserves a balance between its compactness and accuracy of describing the semantics of functions.

• Non-sound analysis: by abandoning soundness, the analyzer improves its accuracy and performance.

• Analysis within one function is based on value analysis. The analyzer tracks values of variables and memory cells and associates the majority of properties with values of variables.

• All analyzes are carried out simultaneously. Each individual checker has low computational cost.

SvEng is designed to detect defects of various types: null pointer dereference, unreachable code, buffer overflow, memory and resource leaks, improper use of library functions, and double mutex lock.

We implement the procedure for finding taint vulnerabilities in such a way as to take full advantage of the analyzer's capabilities. Checkers are implemented as plugins based on the source–sink scheme, where the sources are functions that return tainted data and

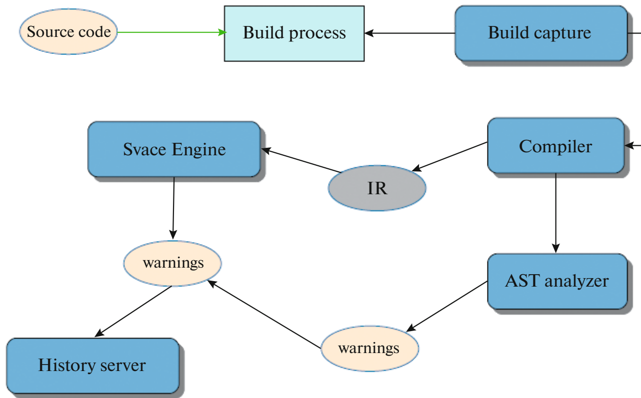


**Fig. 2.** Analysis scheme.

the sinks are operations in which these data must be checked before use. The analyzer is good at detecting errors when the source and sink are not very distant from one another in the call graph.

The analysis is carried out based on the following scheme:

(1) files with intermediate representation are input to the analyzer;

(2) a call graph is constructed;

(3) a preliminary phase (where lightweight analyzes are available) is carried out; during this phase, among other things, information about function pointers and virtual calls is collected;

(4) the call graph is completed for virtual functions and calls by pointers;

(5) a main phase is carried out.

### *4.2. Intermediate Representation*

The analyzer uses its own intermediate representation (Svace IR), which is built upon launching the analyzer. The analyzer receives input files in the following formats depending on the language:

- LLVM files for C/C++;
- Java bytecode for Java/Kotlin;
- JSON format for Go.

Upon converting the source code of a program to the Svace IR, the analysis for all programming languages is carried out in the same manner.

The Svace IR is a low-level typed language in SSA form. The language has procedures and can call them by pointers. To model virtual calls in C++, virtual tables are explicitly constructed. To model exceptions, the goto statement is used.

Listing 3 contains an example of C code, with the corresponding Svace IR snippet shown in Listing 4.

The use of the low-level intermediate representation has both pros and cons. The disadvantages are as follows:

- the analysis of high-level constructs is complicated;
- there is no information about original syntactic constructs;
- transformation of the representation into an equivalent one by the compiler.

The main advantage is the simplicity of the analysis. The semantics is accurately modeled by the compiler; the language has a small number of instructions, which rarely change. A significant number of language constructs are syntactic sugar (exceptions, constructors and destructors, loops, etc.), which does not require adding new instructions to the intermediate representation.

The low-level intermediate representation seems to be a good choice for taint analysis because, for taint-based defects, the very possibility of exploiting the vulnerability, which does not change when switching to a low-level language, is important. In this case, analysis of syntactic constructs is not very useful.

### *4.3. Interprocedural Analysis*

We employ summary-based interprocedural analysis.

In this case, a summary (short description of function's behavior) is generated for each function. The summary is used to analyze the function call statement and allows one to avoid re-analyzing the body of the function. The summary is created upon analyzing the function.

Figure 3 shows a call graph for two programs. At the top of the graph are the main functions, which call other functions. Functions h and j are the leaves of the call graph; the analysis starts from the leaves. Once they are analyzed, their summaries become available, and the analysis can proceed to function foo.

The analysis has both high speed and good scalability. The latter is achieved due to the fact that the summary is quite compact and does not include all details of the function's behavior. The size of the summary can be limited. In this case, the summary does not describe all effects of a function call; however, the time of analyzing the function call becomes constant.

Due to its advantages, this approach is quite popular and is implemented in many static analysis tools: Prefix [13], Saturn [14], Calysto [15], and CSharpChecker [16].

### *4.4. Preliminary Phase*

When using summary-based analysis only, each function is traversed a limited number of times. At the same time, certain analyzes require information about the entire program or about functions that have higher positions in the call graph.

```
int calc(int f)
{
    int a, b;
    int*p;
    a = 1;
    b = 2;
      p  =  f  ?
&a : &b;

    int x = *p;
    *p = 3;

    int y = *p;

     return x +
y;
}
```

**Listing 3.** C code.

To obtain this information, the preliminary phase is designed, which includes analyzes of the following types:

- analysis of value assignments to function pointers;
- analysis of virtual table fills for C++;
- analysis of class hierarchy for Java;
- value analysis of global variables.

As input, these analyses receive information about global variables and instructions for each function. The analysis is carried out in parallel for different modules.

It does not take into account the order of these instructions; however, the dominant instruction is always analyzed first. Thread-insensitive analysis is chosen so as not to slow down the main analysis too much, because the main properties are processed during the main phase.

The purpose of these analyses is to obtain the necessary information without wasting a significant amount of time.

### *4.5. Devirtualization*

Devirtualization is aimed at enabling procedure calls for function pointers and virtual tables. A virtual function table is a global variable of structure type with a field, i.e., an array that contains pointers to virtual functions accessed through constant values.

The analysis tracks values of variables, structures, and constant array indices.

For each pointer, all possible records are collected. The result of the analysis for each function–pointer pair is a set of callee functions or an empty set if the analysis could not find any candidates.

The analysis results are used in two ways:

- to complete the call graph by adding edges and
- to process call-by-pointer instructions.

The summary-based analysis needs information about the call graph. Therefore, upon receiving information about virtual functions, the call graph is completed. This procedure is simple: it is sufficient to add an edge where a function **may** be called as a result of a virtual call. When analyzing the function during the main phase, in some cases, this information can be updated. A more accurate analysis in the preliminary phase adds fewer extra edges. As long as the main analysis is able to remove extra edges, they do not affect its accuracy. However, the time of the analysis, as well as memory consumption, can increase because each edge in the call graph imposes certain constraints on a possible traversal of functions and requires a summary.

The main analyzer implements a plugin that tracks possible candidates for each pointer. This plugin retrieves data about candidates for each pointer; then,

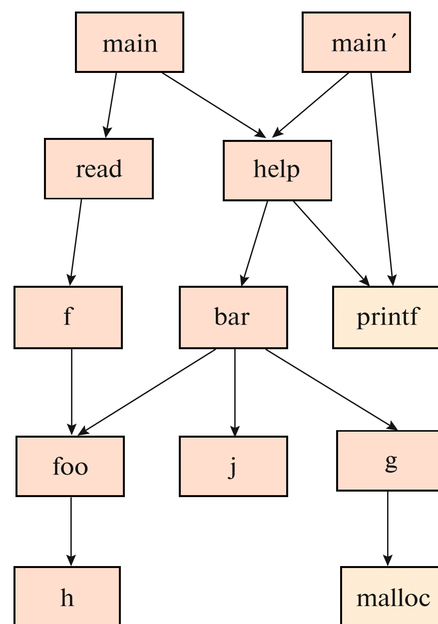


**Fig. 3.** Example of a call graph.

| | |
|---|---|
| <pre>def calc(int f) int<br>{<br>    a = alloca();<br>    b = alloca();<br>     p = alloca ref<br>int();<br>    store 1, a;<br>    store 2, b;<br><br>    if(f != 0)<br>          goto true-<br>label;<br>    else<br>          goto false-<br>label;<br><br>true-label:<br>    store a, p;<br>    goto label2;</pre> | <pre>false-label:<br>    store b, p;<br>    goto label2;<br><br>label2:<br>    t1 = load p;<br>    x = load t1;<br>    store 3, t1;<br>    t2 = load p;<br>    y = load t2;<br>    t3 = x + y;<br>    ret t3;<br>}</pre> |

**Listing 4.** Svace IR code.

using thread-sensitive analysis, it propagates these data independently within the procedure. In some cases, using information about values of variables and unreachable instructions, the analysis can obtain more accurate information than that available after the preliminary phase.

In the case where there are several candidates for a pointer, their conditional call is modeled. For each candidate, a summary is applied; then a union occurs for each context of use. Thus, each summary is applied individually for each context.

### *4.6. Intraprocedural Analysis*

Individual functions are analyzed by symbolic execution with state merging at join points. For a function, a control-flow graph is constructed, and the analysis traverses the graph. Abstract states are associated with the edges of the graph. At each step, an abstract state on the outgoing edge is generated based on an abstract state on the incoming edges.

For strongly connected components (SCCs), several iterations are performed. Abstract states for the outgoing edges of SCCs are saved after each iteration. Once SCCs are analyzed, these states are merged. The analysis uses several heuristics to model all possible paths of the SCC execution. In some cases, the modeling can be incorrect due to the misbehavior of these heuristics.

All additional analyses and checkers run simultaneously, which reduces both analysis time and memory consumption. The analysis time is reduced due to the fact that the properties common to all checkers are analyzed only once. The memory consumption is reduced because, in most cases, once an instruction is analyzed, the analysis state on the incoming edge can be deleted.

To model values of variables and memory cells, Svace uses a special abstraction called the value identifier. Two variables are assigned one value identifier if these variables have the same values at runtime (the value numbering problem is solved).

The majority of properties are associated with value identifiers. The properties themselves can also be described using value identifiers. To describe properties, attributes are used. An attribute defines a property under analysis.

Below are some examples of attributes:

• Null: a pointer has a null value;

• ValueInterval: the values of an integer variable fall within interval [a; b];

• PointsTo: a pointer points to memory cells from a set (memory cells are defined by value identifiers, which model addresses of memory cells);

• Ness: necessary conditions for reaching an edge in the control flow graph; it is a Boolean formula where value identifiers are used as variables.

### *4.7. Using the SMT Solver*

To implement path sensitivity, Svace uses conditional attributes, the properties of which are represented as Boolean formulas over value identifiers. Conditional attributes that describe properties of values of variables are associated with value identifiers.

The formulas are shown in Fig. 4, where *Val* are value identifiers and *Const* are constants. Each literal formula (*Atom*) has a negation statement, which returns a different literal formula. The formulas can use conjunctions, disjunctions from other formulas, and negation for literals.

The SMT solver is run only before issuing a warning to filter out infeasible paths. Therefore, the number of SMT runs does not exceed the number of unfiltered warnings. In the general case, the following formula is used:

$$Ness_i \wedge Error_i(v),$$

where *Ness* is the necessary reachability condition for edge $i$ and $Error_i(v)$ is the error condition for the value modeled by identifier $v$.

**Table 1.** False positive rate for Tizen 6

| Warning | Number of issued warnings | True positive rate |
|---|---|---|
| TAINTED_ARRAY_INDEX | 102 | 62.5 |
| TAINTED_INT | 137 | 65.5 |
| TAINTED_INT.LOOP | 137 | 76 |
| TAINTED_INT.PTR | 82 | 58 |
| TAINTED_PTR | 242 | 70.5 |
| TAINTED.INT_OVERFLOW | 796 | 85 |

### *4.8. Data-Flow Analysis*

To analyze an unreachable code, sound data-flow analysis (DFA)[3] was implemented [17]. It is carried out before analyzing each function in the main phase. The analysis labels unreachable edges in the control-flow graph. The main reason for its implementation was an insufficient accuracy of the main analysis. An unreachable edge strongly affects all other analyses, which is why, in this case, non-soundness can lead to significantly worse results.

Later, the following analyzes were added:

• interval analysis;

• exception analysis;

• live variable analysis.

In addition to sound results, this analysis provides information about a function before initiating the main analysis. With all checkers in the main phase running simultaneously, the problem of preliminary acquisition of properties arises, which is solved by the DFA.

### *4.9. Top-Down Analysis in the Preliminary Phase*

Let us consider an example shown in Listing 5. Function f receives tainted data and passes them to function g. If function g uses its argument unsafely, then a warning must be reported. For this purpose, in the summary-based analysis, the information about the unsafe use of argument y needs to be propagated through the summary.

In the general case, this is not a trivial task. When analyzing function g, there is no information about the contexts of its further use. That is why it is not known whether the information about the unsafe use needs to be stored. The summary is compact, and storing all possible information deprives the analysis of its main advantages.

To solve the problem described above, we supplement the preliminary phase with the DFA. In this phase, procedures are traversed in arbitrary order; tainted data are tracked in the context of a procedure and, for each procedure call in that context, tainted arguments are saved. Thus, the saved information is associated only with the contexts considered.

This approach is somewhat similar to dynamic analysis, whereby the properties of a particular path are investigated with all conclusions being valid only for this path. This analysis investigates properties that are not universal and hold for a certain call context or chain of function calls.

Thus, the preliminary phase gathers information about contexts of use for functions. Data about tainted arguments of functions are used during the main analysis. Attributes that describe tainted data are set based on the results of the preliminary phase. For particular checkers, it looks like an argument is a result of calling a function that returns tainted data. In this case, the problem is that the information about tainted data can get into a summary, which is not desirable. The summary describes the result of a function call for an arbitrary call context, while the preliminary analysis gathers data for certain contexts. We use the following method to solve this problem. A function that has tainted arguments (according to the preliminary traversal) is analyzed twice:

(1) the data from the preliminary traversal are used, and the summary is not created;

(2) the standard analysis without using the data from the preliminary traversal is carried out, and the summary is created.

For instance, if it is known that function `input` is always a source of tainted data, then, when analyzing function `f`, we can conclude that there is a call context for `g` in which its argument has a tainted value. Thus, it is correct to report an error. The main phase makes a decision based only on the analysis of function `g`.

### *4.10. Specifications in Svace*

Specifications are used to analyze library functions the behavior of which is known. A specification in Svace is another definition of a function, which is

[3] In this paper, by the DFA, we mean an engine based on data-flow analysis.

$\bowtie$ ::= > | < | = | ≠ | ≤ | ≥
$\oplus$ ::= + | − | * | /
Op ::= Val | Const
Atom ::= True | False | Op $\bowtie$ Op | Op = Op $\oplus$ Op
Conj = Atom | Conj ∧ Conj
CFormula ::= Conj
SFormula ::= Conj ∧ $\overline{\text{Conj}}$
FFormula ::= Atom | FFormula ∧ FFormula | FFormula ∨ FFormula

**Fig. 4.** Formulas used.

written in a language under analysis. The specification describes the behavior of a function in a compact form. In addition to language constructs, specifications can contain calls of certain predefined functions called special functions.

The distributive of Svace contains specifications for popular libraries. Users can add their own specifications.

For example, let us consider the specification of the `strcat` function from the standard C library. Below is the source code of this specification.

```
char *strcat(char *s, const char *ap-
pend) {
   char d1 = *s;
   char d2 = *append;
   sf_set_trusted_sink_ptr(s);
   sf_set_trusted_sink_ptr(append);
   sf_append_string(s, append);
      sf_vulnerable_fun("This   function
is unsafe, use strncat instead.");
   sf_null_terminated(s);
   return s;
}
```

This code contains the following special functions:

• `void sf_set_trusted_sink_ptr(const void* str` shows that its argument str must be from a trusted source; otherwise, the pointer can cause vulnerability;

• `void sf_vulnerable_fun(const char*const reason)` shows that the current function is not safe and has safe counterparts;

• `void sf_append_string(char* dst, const char* src)` shows that string src has been added to dst;

• `void sf_null_terminated(char *p)` shows that string p ends with a null character.

When analyzing this specification, checkers can extract the following properties:

strings `s` and `append` have been dereferenced, which can be useful for checkers that search for null pointer dereferences;

strings `s` and `append` must be from a trusted source (this information is used by taint checkers);

function `strncat` is vulnerable and its safe counterpart strncat should be used instead;

string s ends with a null character.

## 5. TAINT CHECKERS IN SvEng

### *5.1. Plugins and Checkers*

Analysis in SvEng is divided into the core and plugins. The core analysis tracks the pointer graph, executes strong or weak updates of memory cells, and calls handlers of the corresponding situations. All additional analyses are implemented in plugins as instruction handlers, which operate with value identifiers rather than variables. As input, additional analyses receive an abstract state on the incoming edge of an instruction and form an abstract state on its outgoing edge. Checkers are also implemented as plugins and report warnings based on input abstract states and transfer functions of instructions to be processed.

Information on the incoming edges is available to all additional analyses, whereas information on the outgoing edges is not. Different analyses and checkers must not interfere with each other in the process of instruction analysis. In the current version, analyzers and checkers are run in succession; 4 however, the results should be the same as if they were run in parallel.

In an abstract state, attributes are used to describe properties. Attributes can be associated with value identifiers and with edges of the control flow graph for certain abstract states. An attribute defines a property under analysis. Attributes must have a function that merges two attributes $\sqcup$. This function is used for state merging. Attributes allow abstract states to be shared among additional analyzes.

An attribute can have an arbitrary structure, while attributes of certain types are used quite often. The following types can be distinguished:

• binary attributes;
• ternary attributes;
• interval attributes;
• conditional attributes;
• a set of value identifiers.

Each type can also have a trace: a single linked list of pairs "program point–short text description of an event." Traces change in the process of attribute propagation and are used when reporting warnings to indicate additional points in a program that provide better understanding of an error.

[4] Parallel run of checkers can potentially speed up the analysis; however, it significantly complicates the intraprocedural analysis due to the need for synchronization. That is why parallelization is implemented at the level of the call graph: individual functions can be analyzed in parallel, while the analysis within a function is sequential.

```
void f() {
  int x = input();
  g(x); // the tainted value was passed
}

void g(int y) {
  // context in which tainted value
was passed to variable y is known
}
```

**Listing 5.** Motivation for the preliminary phase.

Binary attributes have two values: true or false. The true value means that some variable surely has an analyzable property, while the false value means either that the variable does not have an analyzable property or there is no enough information. Depending on a merge function, these attributes can be of two types:

- or-attributes: the result is true if at least one argument is true;
- and-attributes: the result is true if both arguments are true.

Ternary attributes can take the following values:

*true*: a property of a variable holds at a given point for all paths5 passing through it;

*maybe*: there is a path, possibly an infeasible one, for which a property holds;

*false*: a property does not hold or there is not enough information.

The attribute merge function is as follows:

- $true \sqcup false = maybe$,
- $maybe \sqcup false = maybe$,
- $true \sqcup maybe = maybe$.

In fact, the ternary attribute is a result of merging the binary or- and and-attributes. Its value is true if both binary attributes are true. Its value is maybe if the or-attribute is true and the and-attribute is not; otherwise, the value is false.

Interval attributes associate a variable with an integer interval that describes an arbitrary property. For instance, it can be a possible amount of memory allocated to a pointer or a value an integer variable can take. The interval can take the following values: $[a,b] - a \le b$; $a$, $b \in [MIN_INT + 1, MAX_INT - 1]$; this means that the property of a variable has a value on the interval from $a$ to $b$. Values *MIN_INT* and *MAX_INT* are reserved for infinities $-\infty$ and $+\infty$.

[5] In the case of static analysis, all paths that the analysis considers feasible are taken into account. The more accurate the analysis, the more infeasible paths it can filter out.

A chain of intervals represents several intervals and allows one to model intervals with excluded points.

Conditional attributes store a formula that describes the fulfillment of a certain property (see 4.7). The satisfiability of the formula is checked by the SMT solver before issuing a warning. This formula consists of the following conjunctions:

- a reachability condition: this formula contains conditions under which a point where a warning is reported is reachable (this formula is stored in attribute *Ness*);
- a taint condition: this formula contains a condition under which a pointer or a value of an integer variable comes from an untrusted source; this formula is tracked by attributes *TaintedPtrIf* (for an unsafe pointer) and *TaintedIntIf* (for an unsafe integer value), which are described below;
- an additional property that depends on a checker, e.g., to access an array, it is checked whether the index value is less than zero or greater than the size of the array.

### *5.2. Interprocedural Attribute Propagation*

When creating a summary, the analysis core determines which value identifiers are included in the summary and calls the *annotate* handler for each of them. When the summary is applied, the analysis core matches formal arguments with actual arguments and calls the *apply* handler. To make interprocedural attributes, it is sufficient to subscribe to these two handlers and implement the corresponding logic depending on the semantics of attributes.

For ternary, binary, and interval attributes, the summary is generated by passing properties without modification (*annotate*).

For conditional attributes, the formula is simplified before being passed to the summary:

(1) add a conjunction with a reachability condition;

(2) save all atomic conditions that contain identifiers added by the core to the summary in a special set;

(3) select an untraversed simple condition from the formula: if it is not contained in the list, then convert it to *False*; otherwise, label it as a traversed one;

(4) apply absorption rules ($False \wedge Cond = False$, where *Cond* is a condition) to the resulting formula;

(5) save the result in the summary.

Hereinafter, we assume that all standard attributes are interprocedural ones and use the summary generation mechanism described above (unless otherwise stated).

### 5.3. Used Attributes

Let us describe auxiliary attributes that provide necessary information for taint checkers.

Attribute *MustTaintedInterval* stores an interval of possible values for a tainted variable. Values from this interval must be checked before use. The attribute merge function is the intersection of intervals ($[10, 20] \cap \varnothing = \varnothing$, $[10, 20] \cap [10, 11] = [10, 11]$).

Attribute *MightTaintedInterval* stores the maximum tainted interval. The attribute merge function is the union with a void interval (($[10, 20] \cup [10, 11] = [10, 11]$) and intersection with a non-void interval ($[10, 20] \cup [10, 11] = [10, 11]$).

The following attributes, which indicate whether the value of a tainted variable has been checked, are also associated with the value of the variable:

*MinIsTainted* is a binary or-attribute that indicates the conduction of a lower bound check (by default, its value is *true*, which means that the check has not been carried out);

*MaxIsTainted* is a binary or-attribute that indicates the conduction of an upper bound check (by default, its value is *true*, which means that the check has not been carried out).

These attributes are required to avoid reporting false positives in the case where a variable has been compared with some function parameter:

```
char* allocate(int max) {
    unsigned int n;
    scanf("%d", n);

    if (n > max) {
        printf ("parameter too big, use %d", max);
        return 0;
    }

return malloc(n);
}
```

The attribute suppresses the issuance of warnings in the cases where an exact safe boundary is not known. For instance, this constraint cannot be statically defined to allocate memory from the heap. When accessing an array with a known size, this boundary is known, and the attribute does not affect the issuance of warnings.

### 5.4. Tainted Integers

Values of integer variables can be controlled by a malefactor. All checkers are implemented based on the source–sink scheme, where the sources are functions that receive data from external sources and the sinks are operations where these data must be checked.

The sources are all data received from the outside of a program (file, network, or user input). In most cases, these data in Svace come from specifications. An exception is the argc and argv parameters of the main function. Svace links `argv` with `argc` by using attribute *ArgvVarAttr*, which allows checkers to avoid false positives due to the use of the main's parameters, e.g., when the `argv` pointer is shifted using `argc`.

The sinks are

- use of a variable as an array index (its range must be checked before use);
- library functions;
- use as a loop step or loop constraint;
- use as a pointer index (even though its exact size is not known, this does not mean that any size can be used).

A specific property of tainted integer variables is that checking their range is quite a complex procedure that uses binary arithmetic.

In addition, certain variables can be interrelated by some operation; in this case, only one variable can be checked. When implementing the analysis, it is important to take these relationships into account. For this purpose, the SMT solver is used: formulas are derived to describe relationships between variables; then, the SMT solver is called to determine whether the formula has a model.

Svace implements the following checkers for finding tainted integers:

- TAINTED_ARRAY_INDEX: access to an array by an unchecked index;
- TAINTED.INT_OVERFLOW: a potential integer overflow;
- TAINTED_INT.PTR: access to a pointer by shifting;
- TAINTED_INT: access to a function where its input parameters must be checked;
- TAINTED_INT.LOOP: the use of a variable as a loop constraint or loop step.

The checkers listed above can use the following suffixes:

- .MIGHT: some paths to unsafe use of data do not contain tainted data;
- .COND: the sink is in a callee function and it is not reachable on all paths within that function.

For example, TAINTED_ARRAY_INDEX.MIGHT is an access to an array as an index where some paths do not contain tainted data.

*5.4.1. TAINTED_INT Warning*

There are cases where the programmer uses a variable the value of which comes from an external source (e.g., in functions like `strncpy` and `malloc`) or it is used as a loop termination condition. Since the malefactor can pass any value, the use of these variables can lead to vulnerabilities (infinite loop or array overflow).

```
void test(int fd) {
    int sizeBuf;
    //sizeBuf is received from an untrusted source
    int ret = recv(fd, &sizeBuf, sizeof(sizeBuf), 0);
    if(ret<0)
         return;
    if (sizeBuf < 0) {
        return;
    }
    //use of tainted variable in calloc
    char*x = calloc(1, sizeBuf);//TAINTED_INT
}
```

In this example, the amount of allocated memory depends on the tainted variable: *sizeBuf* can be very large (which results in allocating an excessively large amount of memory that will not be used) or *sizeBuf* can be zero (in this case, an access to *x* can cause an array overflow).

The checker detects when tainted integer variables are passed to functions, causing vulnerability.

To identify tainted integer variables, the *TaintedIntIf* attribute is used, which stores a formula the execution of which forces the variable to have a tainted value. The attribute merge function is the conjunction of formulas from branches.

Attribute *TrustedIntSinkFlag* is used for interprocedural propagation of properties that describe the use of trusted data. In fact, when applying a summary, this attribute creates another sink for which a warning can be reported. It should be noted that analysis of specifications is a special case of interprocedural analysis. This attribute is used when processing specifications for functions like *malloc*.

A warning is reported if,

• upon calling a function, its argument has the following properties: *TrustedIntSinkFlag* is *true* or the variable is used in loop conditions;

• the variable has non-empty attribute *MustTaintedInterval* or *MightTaintedInterval*;

• the lower and upper bounds of the variable have not been checked; attributes *MinIsTainted* and *MaxIsTainted* are *false*;

• the conjunction of a formula from *TaintedIntIf* is satisfiable under the condition that the variable is in

**Table 2.** False negative rate for Juliet

| CWE | Number of tests | Coverage | FN(%) |
|---|---|---|---|
| CWE680 | 384 | 206 | 46.35 |
| CWE194 | 816 | 444 | 45.59 |
| CWE195 | 816 | 444 | 45.59 |
| CWE789 | 384 | 92 | 76.04 |
| CWE127 | 240 | 104 | 56.67 |
| CWE124 | 240 | 104 | 56.67 |
| CWE126 | 390 | 104 | 73.33 |
| CWE400 | 624 | 164 | 73.72 |
| CWE134 | 1200 | 226 | 81.17 |
| Altogether | 5094 | 1888 | 62.93 |

Overall test coverage is 38.32%.

an interval from *MustTaintedInterval* or *MightTaintedInterval*.

The use of *MustTaintedInterval* and *MightTaintedInterval* is not necessary in terms of program logic; however, it allows us to optimize queries to the SMT solver.

#### *5.4.2. TAINTED_INT.LOOP Warning*

It is a subtype of TAINTED_INT for loop vulnerabilities.

This warning is reported in the following two cases.

• A tainted value is used to limit the number of loop iterations. The error is that the loop can have too many iterations. To determine whether the variable limits the number of loop iterations, information about the control-flow graph is used. The handler of conditional statements checks whether the statement belongs to a strongly connected component and whether its incoming edge is an input edge to that component.

• A tainted value is used as a loop step. In this case, if the malefactor manages to set a value such that the variable remains constant on different iterations, then an infinite loop occurs. The implementation of the loop step detection is more complicated. At the first stage, based on the DFA, loop invariants (variables that have the same values at all iterations) are computed. For the other variables, it is checked whether they are used in arithmetic instructions and whether their ranges permit undesirable values[6] and conditional statements that check the values of these variables.

There are cases where a variable is used in a loop in a callee function. The analysis uses attribute *LoopBoundFlag*, which is set to true in the cases where the variable is considered a constraint on the number of loop iterations. Then, this attribute is propagated interprocedurally; if, at the time of applying a summary, a formal argument has this attribute, while an actual argument has attribute *MustTaintedInterval*, then a warning is issued.

#### *5.4.3. TAINTED_INT.PTR Warning*

If a tainted integer variable is used as a pointer offset without any checks, then allocated memory can be exceeded because the tainted variable can have arbitrary value.

[6] Generally, it is zero. In some cases, integer overflow can also cause an error.

```
void test(int fd, int *ptr) {
    int index;
    //value of index is tainted
    int ret = recv(fd, &index, sizeof(index), 0);
    //use of tainted index as an offset
    ptr[index] = 3;//TAINTED_INT.PTR
}
```

In this example, variable *index* can have any value, which can cause a buffer overflow.

The checker finds situations where tainted integer variables are not checked and are used as pointer offsets.

A warning is issued if

• a pointer access instruction is executed;

• the offset has non-void *MustTaintedInterval* or *MightTaintedInterval*;

• the lower and upper bounds of the offset have not been checked; attributes *MinIsTainted* and *MaxIsTainted* are *false*;

• the amount of memory allocated to the pointer is not known or *MustTaintedInterval* or *MightTaintedInterval* can exceed its size.

#### *5.5.4. TAINTED_ARRAY_INDEX Warning*

This warning is quite similar to TAINTED_INT.PTR; the difference is that it is issued when processing arrays the sizes of which are known to the static analyzer.

```
void test(int fd) {
     int ptr [6];
   int index;
   //value of index came from an untrusted source
   int ret = recv(fd, &index, sizeof(index), 0);
   //use of tainted value as index
   ptr[index] = 3;//TAINTED_ARRAY_INDEX
}
```

In this example, the size of array *ptr* is known, and the value of *index* can exceed 5.

The checker finds situations where an array is accessed by an index received from an untrusted source and the value of this index can exceed the size of the array. In this case, the analyzer knows the size of the array.

For this purpose, it uses attribute *BufferSizeAttrVal*, which stores a possible size of the array as an interval.

A warning is issued if

• the array is accessed by index;

• the size of the array is known; the interval from *BufferSizeAttrVal* is not empty and not $[-\infty,+\infty]$;

• the index has non-empty *MustTaintedInterval* or *MightTaintedInterval*;

• for the index, a formula is derived with the condition that its value can exceed the interval from *BufferSizeAttrVal*; this formula is satisfiable.

#### *5.4.5. Integer Overflow*

The value of a variable that comes from an external source can be arbitrary. If arithmetic operations with this variable are carried out without preliminary check, then an integer overflow can occur and its value will be incorrect. This can lead to both vulnerabilities and violation of program logic. In these cases, the TAINTED.INT_OVERFLOW warning is issued.

```
void test(int fd) {
     int ptr [6];
   int index;
   //value of index came from an untrusted source
   int ret = recv(fd, &index, sizeof(index), 0);
   //integer overflow
   index += 1; //TAINTED.INT_OVERFLOW
   if(index > 4) {
      return;
   }
   ptr[index] = 3;
}
```

In this example, *index* can have the maximum value for the *int* variable; hence, with incrementation, an integer overflow can occur and its value will be incorrect.

The checker finds situations where a variable from an untrusted source is used in arithmetic operations (addition, multiplication, and subtraction). To check the possibility of the overflow, the interval from attribute *MustTaintedInterval* is used.

A warning is issued if

• the addition, multiplication, or subtraction instruction is executed;

• attribute *MustTaintedInterval* of one of the arguments in this instruction is not empty;

• the interval is *MustTaintedInterval* $= [-\infty;+\infty]$ or it can overflow the type of the second argument.

### *5.5. Tainted Strings*

Sometimes, it is required that a pointer contain checked data, e.g., when opening a file by using the open statement or when concatenating or copying strings. In the case of `strcpy` and `strcat`, if string `src` is tainted, then its length can exceed that of string `dst`, causing a buffer overflow.

```
void test(int fd, int *ptr, int size) {
    //env came from an untrusted source
    char* env = getenv("PATH");
    char *buf = malloc(size);
    //string in env can have arbitrary size
    //which can cause buf overflow when copying
    strcpy(buf, env);//TAINTED_PTR
}
```

In this example, the string in env can have arbitrary size, which can cause a buffer overflow when copying it to `buf`.

The checker finds situations where an unsafe pointer is used in functions.

To identify a tainted pointer, the following attributes are used.

*TaintedPtrIf* stores the conditions under which the pointer contains tainted data. This attribute is a formula of propositional logic. The attribute merge function is the conjunction of formulas from branches.

*TaintedPtr* is a ternary attribute that indicates whether the pointer contains tainted data.

Using ternary attribute *TrustedPtrSinkFlag*, the checker finds pointers used in unsafe functions (`open`, `strcpy`, `strcat`, etc.) where a tainted pointer can cause vulnerability.

A warning is issued if

- attribute *TaintedPtr* is *true* or *maybe*; the pointer is surely or possibly received from an untrusted source,
- attribute *TrustedPtrSinkFlag* is *true*, the pointer is used in a function where it can cause vulnerability,
- the formula in *TaintedPtrIf* is satisfiable.

There are also cases where a tainted pointer cannot cause vulnerability, e.g., in the case of using `strcpy` when it is known that there is enough memory allocated for the `dst` string to copy the tainted `src` string:

```
void test(int fd, int *ptr, int size) {
    //env contains tainted data
    char* env = getenv("PATH");
    //size of buf depends on length of string in env
    char *buf = malloc(strlen(env) + 1);
    //overflow is not possible
    strcpy(buf, env);//TAINTED_PTR
}
```

In this example, there is enough memory allocated for the *buf* array to copy the *env* string.

To filter out these situations when copying a string, the checker recognizes the identifier that determines the amount of memory allocated to the string; then, it checks the lengths of which strings are contained in this identifier: if there is a `dst` string among them, then a warning is not reported. In this example, (*strlen*(*env*) + 1) bytes of memory are allocated to variable `buf`, which includes the length of the env string, so a warning is not generated.

The situation with `strcat` is similar. The difference is that, when adding a new string, it is checked whether the amount of allocated memory is sufficient for a `src` string and the set of strings that constitute a `dst` string.

When a tainted string is compared with some other string by using comparison functions (`strcmp`), a warning is also not reported. To identify tainted strings, ternary attribute *SanitizationInvoked* is used. It taints the variables used in string comparison functions.

## 6. RESULTS AND DISCUSSION

### *6.1. Analysis of Open Source Projects*

To estimate the true positive rate, we analyzed Tizen 6 [18], which is a Linux-based open source operating system. The total amount of its source code used for analysis was over 32 million lines. The results are shown in Table 5. For analysis, at least 40 warnings per each type of checkers were reviewed. In this case, the exploitability of errors was not estimated. A warning was considered true if the code contained unsafe

data transfer in critical operations; path feasibility from program entry points was not checked.

The TAINTED.INT_OVERFLOW checker proved quite noisy. It may need some refinement to avoid unnecessary warnings. Most of the found cases are represented as follows:

```
int count;
count = strtol(arg, NULL, base);
```

The *strtol* function returns the *long* type, which is why there is a possibility of loosing a significant portion of the return value.

### *6.2. Juliet 1.3*

The Juliet project [19] is a test suite for static analyzers. It includes both flawed cases (where the analyzer must report warnings) and correct cases (where the analyzer must not report warnings).

From the Juliet set of error types, we took those that can be associated with tainted data: CWE680, CWE194, CWE195, CWE789, CWE127, CWE124, CWE126, CWE134, and CWE400. These tests were compiled with the omitgood option, which hides all tests that does not contain errors. The tests use a special naming system: the name of the test can be divided into parts, each part carrying certain information, e.g., the error type or the source and sink [20]. Basic information about the functions used in the test can be obtained from the functional variant name. From the resulting sample, we filtered out the following tests:

• tests compiled only for Windows; these tests contain w32 and wchar in their functional variant names;

• tests that do not contain tainted data; their functional variant names contain functions rand, new, etc. (rather than tainted sources).

We measured test coverage on the resulting sample. If one of the taint checkers report a warning in a test, then the test was regarded as covered. All non-covered tests were classified as false negatives. Table 2 shows the percentage of non-covered tests.

We also measured the number of false positives on this sample. For this purpose, the tests were compiled with the omitbad option, which hides all tests that contain errors. In this case, any warning was considered false. As a result, the false positive rate was insignificant (0.47%).

## 7. CONCLUSIONS

In this paper, we have described the context- and thread-sensitive interprocedural analysis of tainted data that finds vulnerabilities in C, C++, Java, Kotlin, and Go programs. The analysis uses well-known and well-proven solutions implemented in other tools.

The unique general scheme of the analysis has been developed based on more than 10-year experience in static analysis. The proposed solution does not find all vulnerabilities; however, the percentage of detected errors exceeds 38.32% on Juliet tests.

## 8. FUNDING

This work was supported by the Russian Foundation for Basic Research, project no. 20-01-00581 A.

## REFERENCES

1. Belevantsev, A., Borodin, A., Dudina, I., et al., Design and development of Svace static analyzers, *Proc. Ivannikov Memorial Workshop (IVMEM),* 2018, pp. 3–9.
2. Borodin, A. and Belevancev, 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.
3. Borodin, A., Belevantsev, A., Zhurikhin, D., and Izbyshev, A., Deterministic static analysis, *Proc. Ivannikov Memorial Workshop (IVMEM),* 2018, pp. 10–14.
4. Ivannikov, V., Belevantsev, A., Borodin, A., et al., Svace: Static analyzer for detecting 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.
5. Aleph One, Smashing the stack for fun and profit, *Phrack,* 1996, vol. 7, no. 49, pp. 14–16.
6. National Vulnerability Database, CWE Over Time, 2020. https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time. Accessed January 15, 2021.
7. Landi, W., Undecidability of static analysis, *ACM Lett. Program. Lang. Syst.,* 1992, vol. 1, no. 4, pp. 323–337.
8. Hind, M., Pointer analysis: Haven't we solved this problem yet?, *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Engineering,* 2001, pp. 54–61.
9. Landi, W., Interprocedural aliasing in the presence of pointers, *PhD Thesis,* The State University of New Jersey, 1992.
10. Landi, W. and Ryder, B.G., A safe approximate algorithm for interprocedural aliasing, *ACM SIGPLAN Not.,* 1992, vol. 27, no. 7, pp. 235–248.
11. Livshits, B., Sridharan, M., Smaragdakis, Y., et al., In defense of soundiness: A manifesto, *Commun. ACM,* 2015, vol. 58, no. 2, pp. 44–46.
12. Belevantsev, A., Izbyshev, A., and Zhurikhin, D., Monitoring program builds for Svace static analyzer, *Syst. Admin.,* 2017, nos. 7–8, pp. 135–139.
13. 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.
14. Aiken, A., Bugrara, S., Dillig, I., et al., An overview of the Saturn project, *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Engineering,* 2007, pp. 43–48.
15. Babic, D. and Hu, A.J., Calysto: Scalable and precise extended static checking, *Proc. 30th Int. Conf. Software Engineering,* 2008, pp. 211–220.

16. 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.

17. 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

18. Tizen 6.0 Public M2 Release. https://www.tizen.org/blogs/bighoya/2020/tizen-6.0-public-m2-release-0. Accessed January 15, 2021.

19. Black, P.E., Juliet 1.3 test suite: Changes from 1.2, US Department of Commerce, National Institute of Standards and Technology, 2018.

20. Juliet test suite v1.2 for C/C++ user guide, Center for Assured Software, National Security Agency, 2012.

*Translated by Yu. Kornienko*