ISSN 0361-7688, Programming and Computer Software, 2019, Vol. 45, No. 7, pp. 365–371. © Pleiades Publishing, Ltd., 2019.

Dynamic Detection of Use-After-Free Bugs

S. A. Asryan[b,*], **S. S. Gaissaryan**[a,c,e,f,**], **Sh. F. Kurmangaleev**[a,***], **A. M. Aghabalyan**[d,****], **N. G. Hovsepyan**[d,*****], **and S. S. Sargsyan**[d,******]

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

[b] *Institute of Problems in Informatics and Automation, Armenia National Academy of Sciences, Erevan, 0014 Armenia*

[c] *Faculty of Computational Mathematics and Cybernetics, Moscow State University, Moscow, 119991 Russia*

[d] *Erevan State University, Erevan, 0025 Armenia*

[e] *Moscow Institute of Physics and Technology, Dolgoprudnyi, Moscow oblast, 141700 Russia*

[f] *State University—Higher School of Economics, Moscow, 101000 Russia*

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

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

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

*****e-mail: anna.aghabalyan@ispras.ru*

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

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

Received February 15, 2019; revised February 15, 2019; accepted February 15, 2019

Abstract—A novel method for detecting use-after-free bugs based on the program dynamic analysis is described. In memory unsafe programming languages, such as C or C++, this class of bugs mainly occurs when the program tries to access an area of dynamically allocated memory that has been already freed. For each program execution path, the method checks the correction of the allocation, deallocation, and access operations. Since the dynamic analysis is used, bugs can be found only in the parts of the code that was actually executed. The symbolic program execution with the help of SMT (Satisfiability Modulo Theories) solvers is used. This allows us to generate data the processing of which produces new execution paths.

DOI: 10.1134/S0361768819070028

1. INTRODUCTION

Software often contains bugs of the type

- use-after-free (UAF) and
- buffer or heap overflow.

Since a large part of software is used in critically important applications, bugs can cause serious consequences. There are a number of tools that help resolve this problem using static [1, 2] and dynamic [3–7] analysis.

Static analysis examines the code without executing it. A drawback of this method is that no information about the state of the program during execution (registers, program execution trace, input data, etc.) is available, which results in a large number of false positives. For this reason, static analysis is mostly used prior to the dynamic analysis to reveal fragments of the program that can potentially contain bugs.

The tool described in [1] designed for detecting UAF bugs performs the analysis similar to the analysis of available expressions (the expression $x + y$ is available at the point p if this expression is computed on any path from the entry point to p and x and y remain invariable after the last such computation until p [8]). All paths in the program are examined and the condition *object is defined before its use* is checked. If this condition is not satisfied, the memory use is considered erroneous, and a message is generated.

GUEB [2] is based on the examination of the program binary code. The process of analysis is divided into two main phases. In the first phase, the heap access and address assignment operations are traced (the correspondence between pointers and heap elements). The pairs {address, size} are stored in the sets *alloc_set* and *free_set* when memory is allocated and freed, respectively.

In the second phase, UAF bugs are detected. Using the data collected in the first phase for each point in the program, the tool constructs the set *access_heap* that contains all pairs {address, size} available at this point. If the intersection of the sets *access_heap* and *free_set* is nonempty, then a UAF bug is registered.

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) abort(); // error
}
```

Fig. 1. An example of program from [10].

A reason for the popularity of dynamic analysis is its capability to examine programs at run time, which gives access to registers and memory contents. The tool Avalanche [3] iteratively analyzes the program's execution code based on dynamic binary translation. In the process of analysis, the tool computes the input data of the program in order to automatically traverse all attainable paths in the program and detect its abnormal termination.

In DangNull [4] and FreeSentry [5], the focus is on detecting and nullifying pointers to the dynamically allocated memory after this memory is freed, thus preventing UAF bugs. Both tools use static instrumentation of programs.

Undangle [6] also prevents UAF bugs. This tool labels the return value of each memory allocation function and uses the analysis of tainted data for tracing these labels. When memory is freed, the tool checks which memory locations are associated with the corresponding label and detects dangling pointers (i.e., the pointers that reference an already freed memory location).

The tool Mayhem [7] is based on the method of bug detection in the binary code that combines offline and online approaches to the symbolic execution of programs. The offline approach sequentially examines the program execution paths. During each run, the tool covers only one execution path. A drawback is that the common initial fragment of several paths is repeatedly executed at each run of the program. The online approach examines all possible execution paths simultaneously, which can lead to low memory situations.

The combination of these two approaches works as follows. When the limiting value of the memory consumption is achieved, control points are created, and the examination of some paths is suspended, and data about the current state of the execution, symbolic execution context, and specific input data are saved. When resources become available (the examination of certain paths has completed), one of the control points is recovered (i.e. the execution up to this point is reproduced using the saved data). Next, the symbolic execution context is loaded, and the analysis of a new path begins. This approach makes it possible to avoid the repeated symbolic execution of the program up to the control point.

In this paper, we consider the approach based on the dynamic analysis and dynamic instrumentation [9, 10]. We describe a method for detecting UAF bugs that checks the correctness of using pointers for all possible execution paths of the program. The method is based on the code coverage algorithm used in SAGE [11], and it uses the infrastructure of the dynamic analyzer Triton [10].

In this work, we modified the code coverage algorithm used in Triton, which significantly improved its performance, and we added the support of the analysis of programs that work with input data read from files; this feature was not supported in the implementation of Triton.

In the second section of this paper, we describe the code coverage algorithm used in Triton and in the proposed modification. In the third section, we discuss the initial implementation of UAF bug detection and its combination with the dynamic code coverage. The results are presented in the fourth section.

2. THE CODE COVERAGE ALGORITHM

2.1. Code Coverage in Triton

In this paper, we use the algorithm for maximizing code coverage developed in Microsoft and used in SAGE [11]. This algorithm is partially implemented in Triton. It consists of two phases:

• the choice of initial input data and gathering constraints for each program execution path;

• generating new input data by solving logic expressions consisting of constraints gathered in the first phase.

Consider the example of a program shown in Fig. 1.

In order to examine all paths of this program, it must be provided with the input string *bad!*. To produce the required data, the algorithm starts by running the program with the initial input string, which is placed in the list of input data. After the first run, the set of constraints $< i0 \neq \text{b}, i1 \neq \text{a}, i2 \neq \text{d}, i3 \neq \text{!} >$, where $i0$, $i1$, $i2$, and $i3$ are the memory locations *input* [0], *input* [1], *input* [2], and *input* [3], respectively, is obtained.

In the course of the algorithm execution, these constraints are solved to generate, for each element in the list of input data, descendant data satisfying these constraints; the generated data are placed in the list of input data. The program is again run on each element of this list, and the work of the algorithm is resumed.

This process continues until all elements in the list of input data are examined (the pseudocode of the algorithm is shown in Fig. 3). By applying this algorithm to the program shown in Fig. 1 with the initial input string *good*, we obtain the list of solutions shown in Fig. 2.

Since the work of this algorithm requires the program to be run many times, Triton implements the option of saving the program state. This significantly

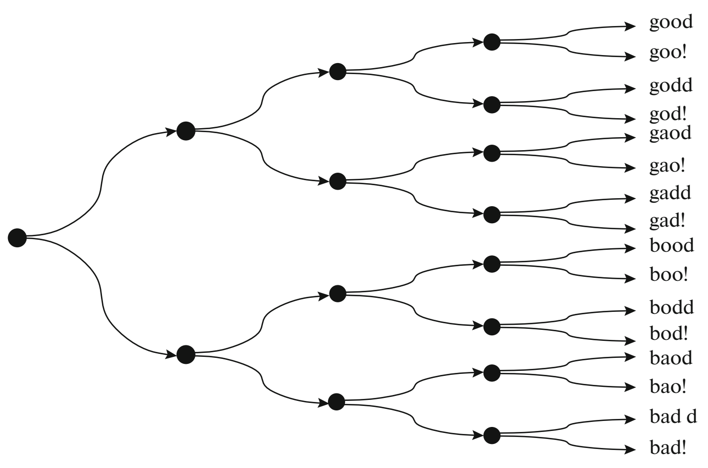


Fig. 2. Input data after each iteration of the algorithm.

```
runCodeCoverage(inputSeed):
  takeSnapshot()
  inputSeed.bound = 0
  inputList = {inputSeed}
  while inputList is not empty:
    input = getInputFromList(inputList)
    convertMemoryToSymbolic(input)
    childInputs = computeNewInputs(input)
    while input childInputs is not empty:
      inputList.append(input)
  if len(inputList)>0 and snapshotEnabled()
      restoreSnapshot()
```

```
computeNewInputs(input):
  // solve constraints using SMT solver
  childInputs = {}
  pc = ComputePathConstraint(input)
  for i in range(input.bound, pc.length):
    if (pc[0..(i-1)]) and not(pc[i]):
      I is solution for pc :
      newIn=updateWithoutOverwrite(input,I)
      newIn.bound = i
      childInputs.append(newIn)
   return childInputs
```

Fig. 3. Pseudocode of the program code coverage algorithm.

improves the performance of the algorithm. Note that a part of the SAGE algorithm [11] designed to decrease the set of input data was not implemented in Triton. For this reason, the tool multiply runs the program under analysis on the input data that do not open new execution paths. In this work, we added new features to the code coverage algorithm that considerably improve the performance.

2.2. Modification of the Coverage Algorithm

In the initial implementation of the SAGE algorithm [11] in Triton, the program always obtained the last element from the list of input data after every iteration without taking into account that the number of program's basic blocks opened using this element. As a result, together with the input data affecting the code coverage, the input data that did not open any new paths were processed.

Since the algorithm generates descendant data of each input element, the length of the list of input data significantly increases. Therefore, in order to execute the algorithm efficiently, the generated input data should be prioritized.

In the algorithm modification proposed in this paper, each element in the list of input data is assigned a weight equal to the number of basic blocks opened by this element. At the start of the algorithm, the input data are assigned a zero weight. During the first iteration of the algorithm, the weights of the initial input data are calculated.

After each iteration, the weights are updated as follows: the weights of the examined elements are passed to the descendant elements (obtained by solving the logic equations). Thus, the hierarchical traversal of input data is used. Before each run of the program, the element with the maximum weight is chosen from the list of input data. This significantly simplifies the list of solutions as is shown in Fig. 4.

It is seen in this figure that after adding weights the amount of input data to be examined decreased almost by a factor of two. This significantly improves the performance of the algorithm (in some tests, the performance increased almost by 90%).

Another drawback of Triton is the support of only the programs that accept only command-line arguments at the input. To extend the range of programs

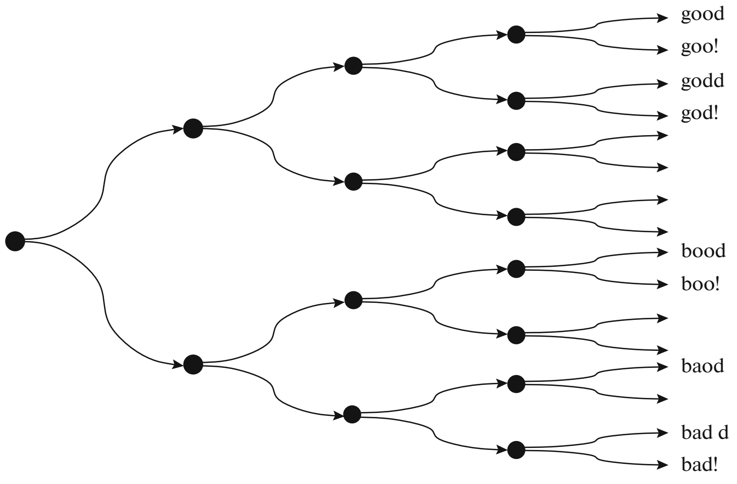


Fig. 4. Input data after each iteration of the algorithm after adding weight.

```
1. char* ptr = (char*) malloc (SIZE);
2. // check status and free ptr
3. if (run_status)
4. free(ptr);
5. // run program
6. if (err)
7. goto exit;
8. return;
9.
10. exit;
11. // double-free: free previously freed memory
12. free(ptr);
13. return;
```

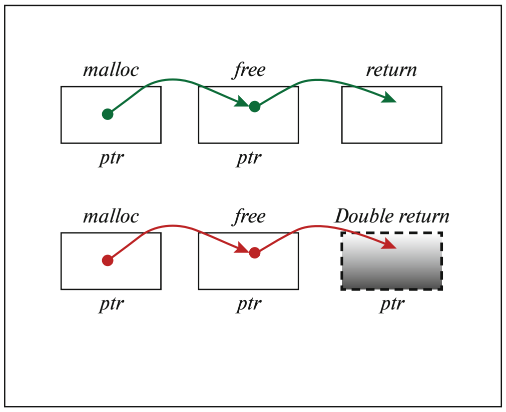


Fig. 5. An example of UAF.

that can be analyzed, we added the support of programs that use files as the source of input data. We also added the capability of determining the ranges of input data that will be marked as symbolic ones in the course of the analysis.

The approach to calculating weights described above is not the only possible one, and in future research other techniques for determining weights will be considered.

3. BUG DETECTION

The dynamic analysis of programs is based on the analysis of programs in the course of their execution. This makes it possible to analyze programs taking into account specific execution conditions and use specific values of pointers. A drawback of dynamic analysis is the requirement to have a good code coverage. However, for the detected bugs, input data on which the bug is reproduced can in many cases be generated. The UAF bug is caused by two successive events:

- creation of a dangling pointer;
- memory access using a dangling pointer.

An example of UAF is shown in Fig. 5. After checking the condition in line 3, the memory referenced by the pointer *ptr* is freed (line 4), and then the control is transferred to line 12, where the same memory location is freed again.

3.1. Triton's Bug Detection Algorithm

Using the program instrumentation, the algorithm traces the memory allocation (*malloc*) and freeing (*free*) functions. At the start of the algorithm, two sets (*allocSET* and *freeSET*) are created. They are used to trace the memory locations that were allocated and then freed during the program execution. The elements of these sets are pairs (*address, size*).

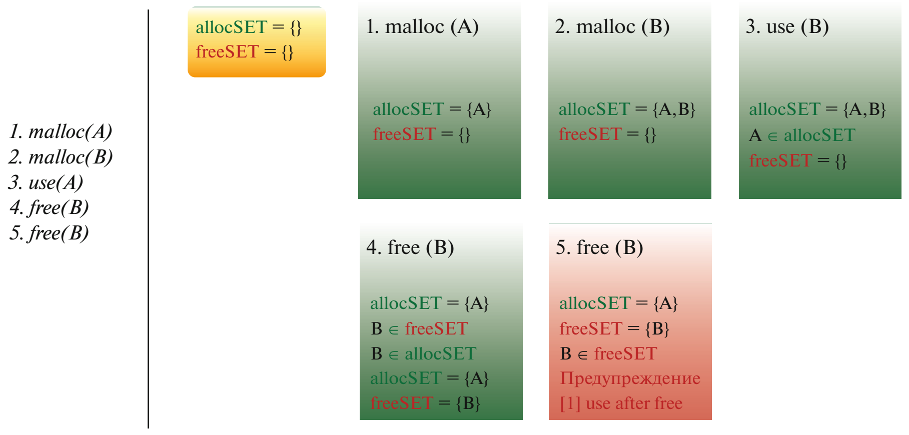


Fig. 6. An example of the algorithm operation.

Each time when *malloc* or *free* are called, the sets *allocSET* and *freeSET* are updated by adding or removing elements with the corresponding address and size of allocated memory. When *malloc* is called, a new element (*address*_2, *size*_2) is created and added to the set *allocSET* and removed from the second set if it is included in *freeSET* (i.e., both the address and size coincide).

If only the address of an element matches, then the address and size are additionally processed before updating the sets (if *size*_2 < *size*_1, then the element (*address*_2 + *size*_2, *size*_*1* – *size*_2) is added to *freeSET*. When the function *free* is called, the corresponding element is moved from the set *allocSET* to *freeSET*.

When memory access instructions are executed, it is checked whether the pointer is in both sets. A UAF bug is registered in two cases: the element is found in *freeSET* but is not included in *allocSet*; one and the same element occurs in *freeSET* more than once. The work of the algorithm is illustrated in Fig. 6.

3.2. The Proposed Method

To improve the efficiency of bug detection, we propose to combine both algorithms described above. The combined method allows one to find UAF bugs on various execution paths that occur due to checks in deeper and nontrivial parts of the program. Figure 7 shows examples of programs in which double deallocation bugs cannot be found without using information about the code coverage (due to the presence of conditional control transfers that will be executed only if the program is run on specific input data).

For the examples shown in Fig. 7, the UAF bug occurs if the program execution reaches lines 21 and 23 in the first and the second program, respectively. The proposed method makes it possible to find the input data that guarantee that the desired block in the code (lines 20–21 and 22–23 in Fig. 7) is reached and then check this part of the code for UAF bugs.

3.3. Comparison of Dynamic Analysis Approaches

Table 1 compares the approach described in this paper with the approach used in Mayhem and Triton.

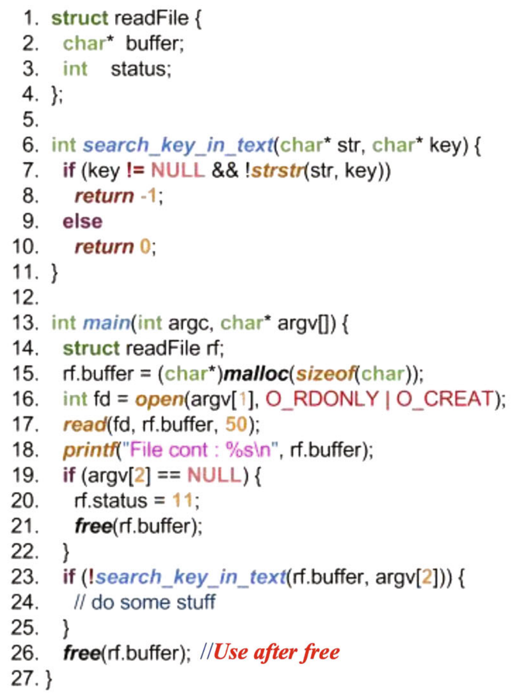
4. RESULTS

The proposed method was tested on manually generated benchmarks including those shown in Figs. 1 and 7; the testing results are presented in Fig. 8. These results show that the performance on manually generated benchmarks increased by about 80% compared with the Triton implementation. The trial runs of the analyzer on real-life programs showed that in the majority of cases the number of symbolic equations is so large that the Triton's code coverage algorithm cannot resolve these equations for all paths.

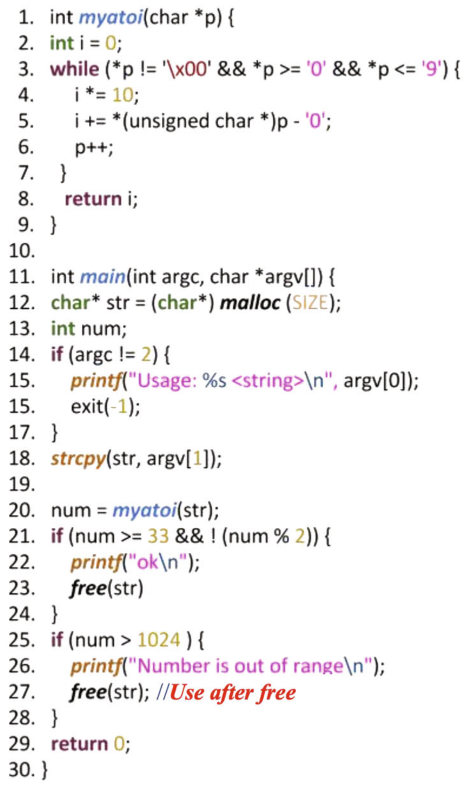
To test the proposed approach, we intentionally injected UAF bugs into the code of real-life projects. We analyzed the projects gvgen from the package

Table 1. Comparison results

	Proposed approach	Mayhem	Triton UAF
Use of code coverage	+	+	–
Offline approach to symbolic execution	+	+	+
Online approach to symbolic execution	–	+	–
Priorities of input data to be processed	+	+	–
Support of files as sources of input data	+	+	–



```
1.  struct readFile {
2.    char*  buffer;
3.    int    status;
4.  };
5.
6.  int search_key_in_text(char* str, char* key) {
7.    if (key != NULL && !strstr(str, key))
8.      return -1;
9.    else
10.     return 0;
11. }
12.
13. int main(int argc, char* argv[]) {
14.   struct readFile rf;
15.   rf.buffer = (char*)malloc(sizeof(char));
16.   int fd = open(argv[1], O_RDONLY | O_CREAT);
17.   read(fd, rf.buffer, 50);
18.   printf("File cont : %s\n", rf.buffer);
19.   if (argv[2] == NULL) {
20.     rf.status = 11;
21.     free(rf.buffer);
22.   }
23.   if (!search_key_in_text(rf.buffer, argv[2])) {
24.     // do some stuff
25.   }
26.   free(rf.buffer); //Use after free
27. }
```



```
1.  int myatoi(char *p) {
2.  int i = 0;
3.  while (*p != '\x00' && *p >= '0' && *p <= '9') {
4.     i *= 10;
5.     i += *(unsigned char *)p - '0';
6.     p++;
7.   }
8.   return i;
9.  }
10.
11. int main(int argc, char *argv[]) {
12. char* str = (char*) malloc (SIZE);
13. int num;
14. if (argc != 2) {
15.    printf("Usage: %s <string>\n", argv[0]);
15.    exit(-1);
17. }
18. strcpy(str, argv[1]);
19.
20. num = myatoi(str);
21. if (num >= 33 && ! (num % 2)) {
22.    printf("ok\n");
23.    free(str)
24. }
25. if (num > 1024 ) {
26.    printf("Number is out of range\n");
27.    free(str); //Use after free
28. }
29. return 0;
30. }
```

Fig. 7. Double deallocation bugs. The first example is based on on the UAF bug in the program libssh. In the first example, the bug can occur in line 26, and in the second example in line 27.

graphviz, jasper from the package libjasper-runtime, and gif2rgb from the package giflib. In these projects, bugs were found at different injection levels. In the case of gvgen, bugs were injected in a number of functions that call each other (the maximum injection depth was three levels (functions)). The injected code was a conditional expression depending on the input data and the code of the bug itself. The satisfaction of this condition resulted in the occurrence of the bug.

In the projects jasper and gif2rgb, injection was done only in one function due to the complexity of the symbolic equations. The injected code was exactly the bug code. We also selected specific code fragments in real-life programs that contained UAF bugs on which the proposed approach was able to find bugs.

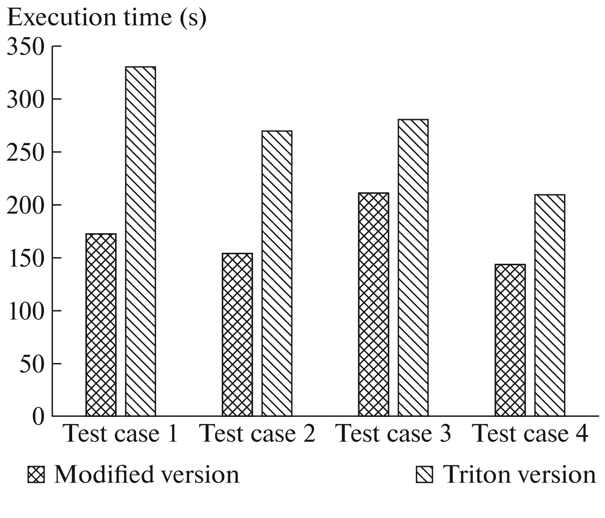


Fig. 8. Comparison results in terms of the analysis execution time.

5. CONCLUSIONS

A method for detecting the use-after-free (UAF) bugs occurring due to incorrect processing of dynamic memory pointers is described. The method was implemented using the Triton infrastructure [10] based on the algorithm described in [11] and the UAF bug detection algorithm. The modification and improvement of the existing implementation allowed us to significantly improve the performance of the analysis.

FUNDING

This work was supported by the Russian Foundation for Basic Research, project no. 17-01-00600.

REFERENCES

1. Dewey, D., Reaves, B., and Trainor, P., Uncovering use-after-free conditions in compiled code, in *Proc. of the 10th International Conference on Availability, Reliability and Security (ARES),* 2015, pp. 90–99.
2. Feist, J., Mounier, L., and Potet, M.-L., Statically detecting use after free on binary code, *J. Comput. Virology Hacking Techniques,* 2014, vol. 10, no. 3, pp. 211–217.
3. Isaev, I.K., Sidorov, D.V., Gerasimov, A.Yu., and Ermakov, M.K., Avalanche: Application of dynamic analysis for the automatic detection of bugs in programs that use network sockets, *Trudy Inst. Sist. Program. Russ. Akad Sci.*, 2011, vol. 21, pp. 55–70.
4. Lee, B., Song, Ch., Jang, Y., and Wang, T., Preventing use-after-free with dangling pointers nullification, in *Proc. of the Network and Distributed System Security Symposium,* 2015. https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/preventing-use-after-free-dangling-pointers-nullification/
5. Younan, Y., FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers, in *Proc. of the Network and Distributed System Security Symposium,* 2015. https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/freesentry-protecting-against-use-after-free-vulnerabilities-due-dangling-pointers/
6. Caballero, J., Grieco, G., Marron, M., and Nappa, A., Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities, in *Proc. of the 2012 International Symposium on Software Testing and Analysis,* 2012, pp. 133–143.
7. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, A., and Brumley, D., Unleashing MAYHEM on binary code, in *Proc. of the 2012 IEEE Symposium on Security and Privacy,* 2012, pp. 380–394.
8. Aho, A., Ullman, J., Sethi, R., and Lam, M. S., *Compilers: Principles, Techniques, and Tools*, Boston: Addison Wesley, 2006, 2nd ed.
9. Pin – A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
10. Triton – Dynamic Binary Analysis Framework. https://triton.quarkslab.com/
11. Godefroid, P., Levin, M. Y., and Molnar, D. Automated whitebox fuzz testing, in *Proc. of the Conference on Network and Distributed Systems Security (NDSS'2008),* 2008, pp. 151–166.
12. de Moura, L. and Bjørner, N., Z3: An efficient SMT solver, in *Proc. of the 14th International conference on Tools and Algorithms for the Construction and Analysis of Systems,* 2008, pp. 337–340.

Translated by A. Klimontovich