Mathematical Problems of Computer Science 48, 50--56, 2017.

# Multiplatform Use-After-Free and Double-Free Detection in Binaries *

Grigor S. Keropyan, Vahagn G. Vardanyan, Hayk K. Aslanyan, Shamil F. Kurmangaleev and Sergey. S. Gaissaryan

Ivannikov Institute for System Programming of the RAS
e-mail: {goqor, vaag, hayk, kursh, ssg}@ispras.ru

**Abstract**

Use-after-free (UAF) defects are a class of memory corruption bugs, which occur when a program continues to use a pointer after it has been freed. Double-free (DF) defects arise when the same memory is freed more than once. The developed platform is capable to analyze binaries of several architectures (x86, x86-64, MIPS, POWER-PC, ARM) and is based on program static analysis approach. For program analysis SDG (System Dependence Graph) machine-independent representation is used. SDG combines call graph, control and data flow graphs of the program. The tool consists of two main components: SDG generation and analysis of the obtained SDG. SDG generation is implemented using Ida Pro [1] disassembler and Binnavi [2] static analysis platform. Experimental results prove the scalability and effectiveness of the developed framework. The tool is tested on several test suits such as Juliet [3]. It also has detected a number of well-known bugs in real-world projects .

**Keywords:** Binary static analysis, Use-after-free, Dangling pointer detection.

## 1. Introduction

Detecting bugs in software is an important task to help to improve security and reliability. It can be done at any time during the software lifecycle. Ideally, all bugs are found during the testing phase before the software is deployed. However, software testing does not find all possible bugs at any given time. Moreover, when a programmer uses memory unsafe languages, such as C/C++, these bugs can lead to security violations. Despite this, these programming languages are still popular because of high performance. A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer. UAF defect arises in case of the usage of dangling pointer and DF defect arises in case of deleting (or freeing) it.

There is a line of research specifically focusing on detecting UAF defects. In general, they can be detected through both static and dynamic analyses. Static analysis methods are often

* The paper is supported by RFBR grant 18-07-01154

based on the disassembly process. Dynamic analysis methods try to trace the program execution and search for memory state to carry out the analysis.

In this paper, we introduce a new approach for detecting UAF defects which supports multiple target architectures and is based on graph representation of program. At first, a system dependence graph (SDG) is generated, which combines the call graph, interprocedural data dependences, the control flow graph and the data flow graph. The background for SDG generation is Binnavi static analysis platform and REIL (the reverse engineering intermediate language) representation [14]. Finally, the main algorithm analyzes SDG for detecting UAF and DF defects. Fig. 1 represents the main architecture of the proposed tool.

## 2. Related Work

Several works are dedicated to source code analysis. N. Nethercote develops an instrument named memcheck using Valgrind [4] which can check all of the memory read / write operations and the interception of calls like malloc, new, free, delete. It is good for detecting memory management problems, but cannot deal with aliasing problems. cppcheck [5] can detect possible memory errors, and provides a simple analysis such as mismatching allocation and deallocation. More detailed analysis is done by Polyspace [6] and Frama-C [7]. The platforms gather several analysis techniques into a single collaborative extensible framework. These tools are mainly focused on safety verification and have several plugins for detecting security violations.

Usually the source code is not available and a binary analysis is required. Additionally, the binary analysis can detect bugs, which do not exist in source code and are generated by compiler during various transformations.

Dynamic analysis solutions ( [8], [9], [10]) for detecting UAF defects generally introduce a high performance overhead, and may miss lots of bugs because they are only able to explore a small portion of the program execution space.

In [11] the authors represent the tool GUEB for static finding of UAF defects in binary files. At first, they track heap operations and address transfers, taking into account aliases, using a value analysis. Then they use these results to statically identify the UAF defects. Finally, they extract subgraphs, for each UAF, describing sequentially where the dangling pointer is created, freed and used. However, the whole process of detecting UAF is not automatic. For instance, the user should manually identify all wrappers to free/malloc functions, which, mainly are not applicable for large binaries. S. Cesare represents an approach [12] for detecting UAF defects using decompilation and data flow analysis. As decompilation is not sound, program behavior may be underapproximated leading to false negatives. Another tool, which is based on static analysis, is represented by D. Dewey et al. [13]. They use gen-kill analysis for detecting available expressions, then they use data flow analysis for determining UAF defects. Dataflow analysis is implemented on small number of x86 assembler instructions, hence their tool works only on x86 binaries.

The suggested approach in the article makes possible to detect UAF and DF bugs in binaries of several architectures. Unlike GUEB it does not need user interaction. The tool does not require high performance overhead and detects defects with 30% false positive rate in average.

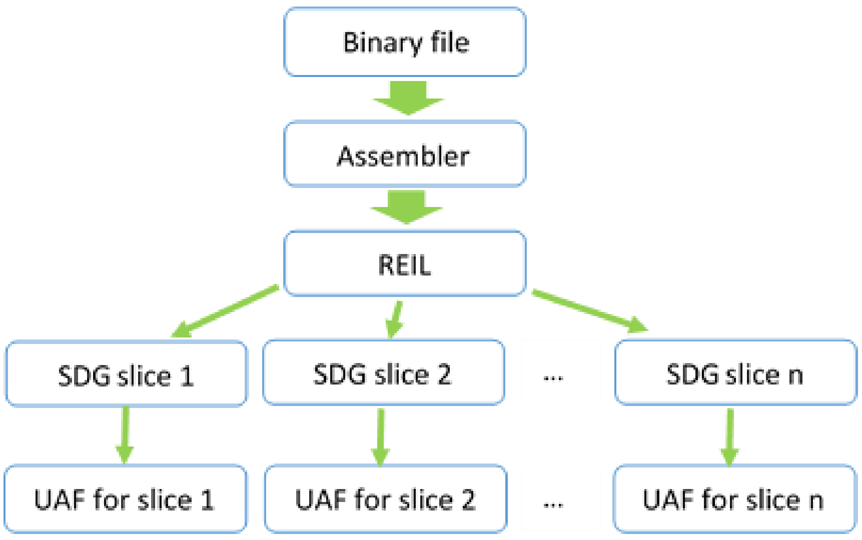

Fig.1 Tool architecture.

### 3.1 SDG Generation

System dependence graph is one of the most detailed structures for representing semantics of the program. It combines call graph, interprocedural data dependences, control and data flow graphs. At the first stage of SDG generation target program is disassembled using IDA Pro. This allows to import disassembled binary into Binnavi platform. Binnavi platform is used to recover the structure of the target program, generate data, control and interprocedual dependencies between instructions and translate program to REIL representation. REIL representation is independent of target architecture and allows to analyze binaries from different architectures such as x86, x86_64, ARM, MIPS and PowerPC. Furthermore, REIL representation has only 17 atomic instructions and only two instructions for memory access (*stm*: store to memory, *ldm*: load from memory) which allows to simplify the analysis process. At first stage of SDG generation algorithm tries to find all free functions of the program. Then the algorithm tries to slice program into independent pieces. Slicing of the program is performed on the call graph. For each *free* function call, algorithm generates all the paths that are leading to that function. Slicing allows to drop away the significant part of program and perform analysis only on the parts which contain memory free operations. The length of the paths can be configured by the user. Finally, the algorithm generates SDGs for each slice. Vertices of these SDGs correspond to REIL instruction and edges correspond to control dependences, data and interprocedural data dependences between instructions. For each REIL instruction an SDG vertex is constructed. Two vertices are connected if there are data (interprocedural or intraprocedural) or control dependencies between the corresponding REIL instructions (Fig.2). Generated slices may contain many common parts (functions). So, SDG generation algorithm uses dynamic programming approach: information about dependencies for all functions is cached in database and reused during SDG generation for each slice. Another big advantage of the program slicing approach is the possibility to organize whole analysis parallel on each independent slice. This is especially important while analyzing big binaries (over 30 MB) where the size of the recovered graphs can grow enormously huge.

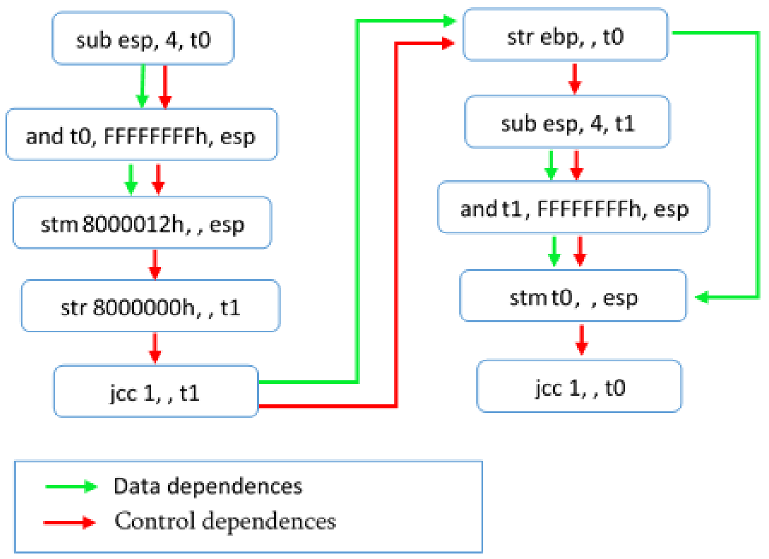

Fig.2 Slice example.

## 3.2 Detecting UAF and DF

The main analysis for detecting UAF and DF bugs is performed on generated SDG slices. Root of these slices is the call to the system *free* function. We start analysis from the system *free* function and try to find freed pointer (argument of *free* function). It can be easily done by using the existing data dependences. The difficulty arises when the freed pointer may have several aliases. Performing alias analysis on the whole program can bring huge performance overhead. In our approach, we have implemented a simple analysis on SDG which tries to find all aliases for the given pointer within a function using available data and control dependencies. There are several possible cases for handling found freed variable:

a. Freed pointer is defined in the function, which contains call to *free* function.
b. Freed pointer is in global scope.
c. Freed pointer is passed as an argument to the observing function.

First and second cases algorithm handles by traversing SDG by control dependencies starting from the *free* function call. Traversing by control dependencies allows to find paths (if any) from free to use of the pointer (or one of its aliases). Then, using data dependencies, algorithm checks redefinitions of pointer on each found path. If there are no redefinition on one of the paths, algorithm reports UAF or DF error.

Third case, when the freed pointer is passed to the observing function as an argument, algorithm marks that function as *free* and adds it to the free functions list. Then the whole analysis of finding UAF is repeated until no new *free function* is found.

Using SDG as an intermediate representation makes analysis path sensitive and interprocedual. However, there are some limitations in the current stage of the development. The main limitation is that during dependencies construction we assume that all arguments are passed to the functions by stack. Support of different calling conversation is in active state of development.

## 3. Experimental Results

Suggested tool is tested against several well-known test suits such as Juliet [3]. Results show that we can find all the UAF and DF errors presented in this suit with 5% false positive rate in average. The tool is also tested on dozens of real word programs that contain real UAF and DF bugs. In the Table 1 the list of projects with UAF and DF defects are presented. These defects were admitted and fixed by the projects maintainers in the newer versions. Last three columns of the table contain analysis results of suggested tool. All tests are performed on the server with 20 physical intel xeon 2.3 GHz processors. Analysis time column in the table proves effectiveness of SDG splitting technique for parallel analysis.

Table 1: Analysis result on several well-known projects containing UAF and DF defects.

| **Project name** | **Version** | **Project size** | **Analysis time** | | | **UAF and DF count** | **False positive** |
|---|---|---|---|---|---|---|---|
| | | | 5 cores | 10 cores | 20 cores | | |
| jasper | 1.900.1 | 1 MB | 231s | 177s | 166s | 3 | 0% |
| giflib | 5.1.2 | 50 KB | 13s | 10s | 10s | 1 | 0% |
| libtiff | 4.0.3 | 1 MB | 278s | 160s | 100s | 12 | 33% |
| gnome-nettool | 3.8.1 | 336 KB | 74 | 52 | 48 | 1 | 0% |
| openslp | 1.2.1 | 700 KB | 60s | 56s | 40s | 4 | 50% |
| libssh | 0.5.2 | 700 KB | 190 | 168 | 99 | 19 | 21% |

The presented tool is also capable to find UAF and DF bugs in gnome-nettool and giflib packages of Debian distributive. False positive rate for examined programs is competitive with other well-known techniques [9], [11], [12]. Moreover, the presented tool is multiplatform and can be used for analyzing binaries for different target architectures.

## 4. Conclusion and Future Work

In the paper a binary static analysis tool is presented that detects UAF and DG defects on several modern architectures. The analysis is performed on System Dependence Graph (SDG) of the target program. The proposed algorithm includes two steps: generation of SDGs for program slices where the root of each slice is a “free” function and analysis of generated slices. Alias analysis is performed on each graph to avoid performance overhead of finding aliases in the whole program. The approach makes it possible to perform the analysis in parallel for each independent slice.

Development and improvement of the tool is in active state. We plan to add support for different calling conventions and continue testing on different real-world programs and libraries to reduce false positive rate and improve quality of analysis.

## References

[1] [Online]. Available: www.hex-rays.com/products/ida

[2] [Online]. Available: www.zynamics.com/binnavi.html

[3] [Online]. Available: www.samate.nist.gov/SRD/testsuite.php

[4] N. Nethercote, "Dynamic Binary Analysis and Instrumentation", PhD Dissertation, University of Cambridge, 2004.

[5] [Online]. Available: www.cppcheck.sourceforge.net

[6] [Online]. Available: www.mathworks.com/training-schedule/polyspace-code-prover-for-cc-code-verification.html

[7] P. Cuoq, F. Kirchner, N.Kosmatov, V. Prevosto, J. Signoles and B. Yakobowski, "Frama-C—a software analysis perspective," *SEFM,* pp. 233-247, 2012.

[8] W. Xu, D. C. DuVarney and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of C programs," *ACM SIGSOFT Software Engineering Notes,* vol. 29, pp. 117-126, 2004.

[9] J. Caballero, G. Grieco, M. Marron and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," *Proceedings of the 2012 International Symposium on Software Testing and Analysis,* pp. 133-143, 2012.

[10] B. Lee, C. Song, Y. Jang and T. Wang, "Preventing Use-after-free with Dangling Pointers Nullification," in *NDSS Symposium 2015,* pp. 17-32, 2015.

[11] J. Feist, L. Mounier and ML. Potet, "Statically detecting use after free on binary code," *Journal of Computer Virology and Hacking Techniques,* vol. 10, no. 3, pp. 211-217, 2014.

[12] S. Cesare, "Detecting bugs using decompilation and data flow analysis," in *Black Hat USA,* 2013.

[13] D. Dewey, B. Reaves and P. Traynor, "Uncovering Use-After-Free Conditions in Compiled Code," in *10th International Conference on Availability, Reliability and Security,* 2015.

[14] www.zynamics.com/binnavi/manual/html/reil_language.htm

**Submitted 02.08.2017, accepted 20.11.2017.**

# Երկուական կոդում ազատված հիշողության օգտագործման և հիշողության կրկնակի ազատման սխալների հայտնաբերման բազմապլատֆորմ համակարգ

Գ. Քերոբյան, Վ. Վարդանյան, Հ. Ասլանյան, Շ. Կուրմանգալեև և Ս. Գայսարյան

**Ամփոփում**

Ազատված հիշողության օգտագործման սխալները հիշողության աշխատանքի հետ կապված սխալների դաս է, որն առաջանում է, երբ ծրագիրը շարունակում է օգտագործել ազատված հիշողությունը: Մշակված համակարգը հնարավորություն է տալիս վերլուծել տարբեր ճարտարապետությունների (x86, x86-64, MIPS, POWER-PC, ARM) երկուական կոդ և հիմնված է կոդի ստատիկ անալիզի վրա։ Ծրագրի անալիզի համար օգտագործվում է SDG (System Dependence Graph) մեքենայից անկախ ներկայացում։ SDG-ն ներառում է ծրագրի կանչերի, ղեկավարման և կախվածությունների գրաֆները։ SDG-ն կառուցվում է՝ հիմնվելով Ida Pro [1] դիսասսեմբլերի և Binnavi [2] ստատիկ անալիզի համակարգի վրա։ Փորձարարական արդյունքները ապացուցում են ներկայացված համակարգի ընդլայնելիությունն ու արդյունավետությունը։ Գործիքը թեստավորվել է բազմաթիվ թեստավորման համակարգերի վրա, որոնցից է Juliet-ը [3]։ Այն նաև հայտնաբերել է բազմաթիվ հայտնի սխալներ կիրառական ծրագրերում։

## Мультиплатформное нахождение ошибок использования памяти после освобождения и повторного освобождения памяти в бинарном коде

Г. Керопян, В. Вартанян, А. Асланян, Ш. Курмангалеев и С. Гайсарян

**Аннотация**

Ошибки использования памяти после освобождения — это класс ошибок памяти, который возникает, когда программа продолжает использовать память после его освобождения. Разработанная платформа способна анализировать бинарные коды нескольких архитектур (x86, x86-64, MIPS, POWER-PC, ARM) и основан на статическом подходе. Для анализа программ используется SDG (System Dependence Graph), независимое от машины представление. SDG объединяет графы вызовов, управления и потоков данных программы. Инструмент состоит из двух основных компонентов: генерации SDG и анализа полученных SDG. Генерация SDG реализована с использованием дисассемблера Ida Pro [1] и платформы статического анализа Binnavi [2]. Экспериментальные результаты доказывают масштабируемость и эффективность разработанного платформа. Инструмент тестировался на разных тестовых наборах, таких как Juliet [3]. Он также обнаружил ряд известных дефектов в реальных проектах.