Multiplatform Static Analysis Framework for Program Defects Detection*

Hayk, Aslanyan
ISPRAS
Moscow, Russia
e-mail: hayk@ispras.ru

Sergey, Asryan
ISPRAS
Moscow, Russia
e-mail: asryan@ispras.ru

Jivan, Hakobyan
ISPRAS
Moscow, Russia
e-mail: jivan@ispras.ru

Vahagn, Vardanyan
ISPRAS
Moscow, Russia
e-mail: vaag@ispras.ru

Sevak, Sargsyan
ISPRAS
Moscow, Russia
e-mail: sevak.sargsyan@ispras.ru

Shamil, Kurmangaleev
ISPRAS
Moscow, Russia
e-mail: kursh@ispras.ru

ABSTRACT
This paper presents multiplatform framework for static detection of the most common program defects occurring due to usage of C/C++ programming languages. The developed platform capable to analyze the source code and binary code of the program. For program analysis, SDG (System Dependence Graph) [1] machine independent representation is used. SDG combines call graph, control and data flow graphs of the program. Vertices of SDG are instructions of the program and the edges are control and data dependencies between these instructions. The framework consists three main components. The first component provides SDG generation from LLVM bitcode. The second component generates LLVM bitcode from binary code. Binary to LLVM bitcode translation is implemented using Ida Pro [2] disassembler and Binnavi framework [3]. Generated bitcode is then used to construct SDG, allowing static analysis on binary code. The third component is the set of static checkers operating on SDG representation. Experimental results prove scalability and effectiveness of developed framework. We have detected number of defects in real world projects. Using developed tool we were able to detect several well-known defects such as CVE-2016-0705 and CVE-2016-0799.

Keywords
Code defects, binary code, LLVM, Static Analysis, SDG.

1. INTRODUCTION
C/C++ are unsafe languages due to possibility to direct memory access via pointers. Despite this, they still popular because of high performance. C/C++ are used for programming operation systems, databases, network protocols, cryptographic libraries, rocket and plain control systems. There are number of instruments for automatic detection of program defects.

A format string is a simple representation of ASCII string in a controlled manner using format specifiers. Further, this complete ASCII string is fed to format functions such as printf, vprintf, scanf to convert the C datatypes into String representation. Format string vulnerabilities in C programs have been studied extensively in recent years. Modern compilers can perform checks during compilation [4]. These approaches mostly based on lexical analysis and do not count semantics of the program. Another approach uses taint analysis to detect format string defects in C program [5]. It annotates the input string as taint and the format string as untaint. The untaint type object can be assigned to the taint one, but not vice versa. If there is a type conflict system reports about format string bug. Although there are many approaches for format string defect detection on source code, analyses on binary code are rare. Most of the works are dedicated to the prevention from format string defect [6], [7]. Currently, our framework detects format strings, which cause using non-correct specifiers.

Use after free vulnerability specifically refers to the attempt to access memory after it has been freed, which can cause a program to crash or lead to the execution of arbitrary code. It is studied in several works. In [8] authors represent tool for finding use after free 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 UAF defects. Finally, they extract subgraphs, for each UAF, describing sequentially where the dangling pointer is created, freed and used. Polyspace [9] and Frama-C [10] work on source level C code. These tools are mainly dedicated to safety verification: programs that do not respect some constraints are rejected, such as undefined behaviors in C.

Another type of bugs is buffer overflow. It is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. The static approaches for finding buffer overflows, consist in performing some code analysis (usually based on data-flow analysis or abstract interpretation), without executing the application. They use taint-dependency analysis [11] (to detect that a user input can be written into a buffer) or more sophisticated value analysis (to detect that a buffer can be accessed out of its bound). Typical existing tools are CodeSurfer[12] and Parfait[13].

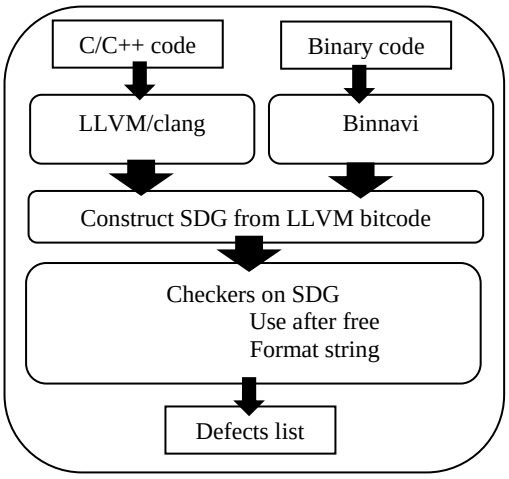
The main purpose of the presented work is to provide common platform for program static analysis. Framework uses SDG as its core representation which provides all necessary information about program semantics. Framework allows to analyze binaries for various target architectures (x86, x86_64, MIPS, ARM, PowerPC) as well as source codes that can be translated into LLVM bitcode. Framework is easily extendable, allowing developers to write own analysis on SDG.

2. THE ARCHITECTURE
Picture 1 represents core architecture of the framework. The first two components are responsible for SDG generation from source or binary code. The third component contains core algorithms for static analysis.

* The paper is supported by RFBR grant 17-01-00600 A Developing methods that protect from vulnerabilities exploitation and also methods for bypassing such protection.

2.2. SDG generation

System dependence graph is one of the most detailed structures for representing semantics of the program. It contains call graph, interprocedural data dependences, control and data flow graphs. SDG is generated based on LLVM bitcode. Picture 2 demonstrates simple *C++* program and the corresponding LLVM bitcode. Appropriate SDG for LLVM bitcode from Picture 2 is shown in Picture 3. For each bitcode instruction an SDG vertex is constructed. Two vertices are connected if there are data or control dependencies between corresponding bitcode instructions (in Picture 3 blue edges represent data dependences, green edges corresponding control dependences).



Picture 1. Architecture of proposed framework

2.1. Translation from Binary to LLVM

In order to enable analysis on binary code, special engine was developed which translates binary code into LLVM bitcode. The engine consists of two main steps. At the first IDA Pro is used to disassemble target binary and store the result in the database. BinNavi [3] is a platform-independent reverse engineering infrastructure designed for binary files analysis. It can be used for new bugs discovery and malware analysis. The most important feature that binnavi provides is control flow based code analysis of x86, x86-64, ARM, MIPS, and PowerPC disassembled code. Binnavi also provides possibility of writing scripts and plugins to extend its functionality for specific goals. Binnavi uses REIL [14] (Reverse Engineering Intermediate Language) as core representation of disassembled files. REIL is a meta-assembly language that is used to write platform-independent analysis algorithms. Structurally it is very simple assembly language, which knows only 17 different instructions. This representation is used to organize translation into LLVM bitcode. First disassembled binary is translated into REIL (this allows to implement analysis for various target binaries, such as x86, ARM, MIPS and PowerPC). After that each REIL instruction is mapped into corresponding LLVM bitcode. Disassembled binaries contain numerous instructions to access program stack. To translate that kind of instructions program stack is emulated in LLVM bitcode. Another challenge was functions calls translation. For now engine supports only cdel default conv convention for argument passing translation. In the future, we plan to extend this functionality to support custom calling conventions. To ensure correctness of translation whole process is tested using compilers tests suits. Comparing to other approaches [15], [16] our method uses REIL representation, making translation independent from architecture of target binary.

```
#include <iostream>
int f(int *p) {
  return *p;
}
int main() {
  int *b;
  f(b);
  return 0;
}
```

```
define i32 @_Z1fPi(i32* %p) #2 {
  %1 = alloca i32*, align 8
  store i32* %p, i32** %1, align 8
  %2 = load i32** %1, align 8, !dbg !21
  %3 = load i32* %2, align 4, !dbg !21
  ret i32 %3, !dbg !21
}
define i32 @main() #2 {
  %1 = alloca i32, align 4
  %b = alloca i32*, align 8
  store i32 0, i32* %1
  %2 = load i32** %b, align 8, !dbg !22
  %3 = call i32 @_Z1fPi(i32* %2), !dbg !22
  ret i32 0, !dbg !23
}
```

Picture 2. Example of LLVM bitcode

2.3. Checkers on SDG

Currently two basic analyses are implemented for SDG. The first analysis detects pointers, which are used after deletion. The second checker detects usages of improperly constructed format strings when using C/C++ standard library functions (e.g printf, sprintf). Both checkers use specially developed taint analysis engine.

2.3.1. Taint analysis

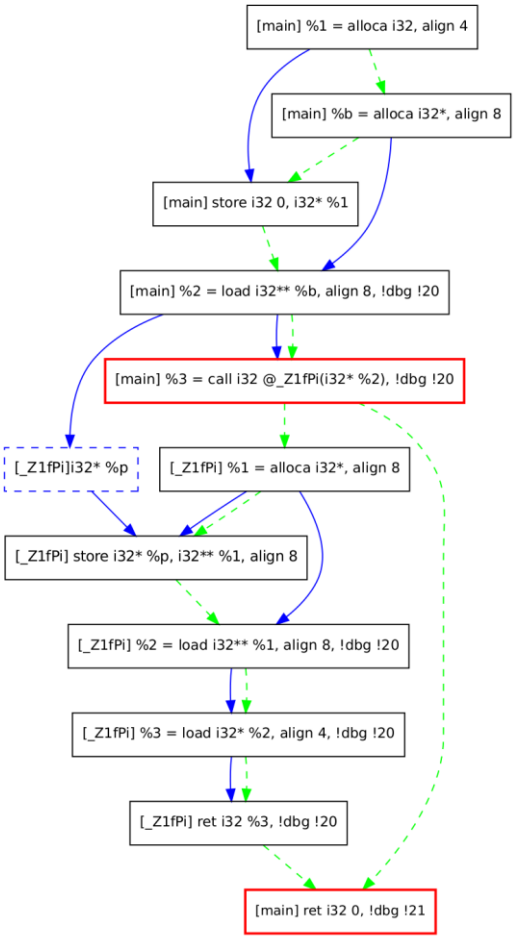
Due to described SDG representation, taint analysis engine has simple implementation. We provide API function for the vertices of SDG, which detects all vertices affected by the tainted one. The function performs BFS by data dependences for given SDG vertex.

2.3.2. Use after free checker

The aim of this checker is to find such pointers, which were used (dereferenced) after, *delete/free* operations, without assigning them another valid pointer. In order to find such cases algorithm at first finds all vertices SDG corresponding to pointers (*[main] %b = alloca i32*, align 8* vertex for the SDG in Picture 1). If for any of detected vertices *P* there are two vertices *D* and *U*, where:

1. Exist data edges path from *P* to *D*.
2. Exist data edges path from *P* to *U*.
3. Exist control edges path from *D* to *U*, where pointer corresponding to *P* was not redefined.
4. Vertex *D* corresponding to delete operation.
5. Vertex *U* corresponding dereference instruction of *P* or is a corresponding delete operation.

Then pointer corresponding to vertex P used after it has been freed (when *U* is a delete operation - double free). Taint analysis engine is used for detection data paths (instructions affecting) from definition of pointers to the use instructions.



Picture 3. SDG example

2.3.3. Format string checker

The main purpose of the checker is detecting strings, which are tainted from the user input and is not correctly used in printf, fprintf, asprintf, sprintf, snprintf, vsprintf, vfprintf, vasprintf, vsnprintf, syslog functions. For any of vertices *F*, which correspond to call instruction one of these functions:

1. If the call instructions do not have format specifiers or their amount is not equal to arguments amount or their types are not appropriate to arguments types, then go to next step, otherwise there is no format string defect.
2. Construct set of vertices corresponding to alloca instructions and have path of data dependences to *F*.
3. Construct set of vertices corresponding to input functions' call instructions and have path of control dependences to *F*.
4. For all vertices I from set of Step 3 of, construct set ALLOCA_INPUT, which correspond to alloca instructions and have path of data dependences to I.
5. Find common vertices of ALLOCA_INPUT and set if Step 2.

If this intersection is not empty, then string variables corresponding to the intersection can possibly be format strings defect.

2.4. The advantages of proposed framework

Proposed framework has three basic advantages:

1. It is applicable for both source and binary that means possibility using on all software development life cycle and prioritize errors which are detected in various levels of representation.
2. The framework is easily extendable. New checkers and platforms can be added easily.
3. The framework scalable for analysis million lines of source code and hundred MBs of binary code.

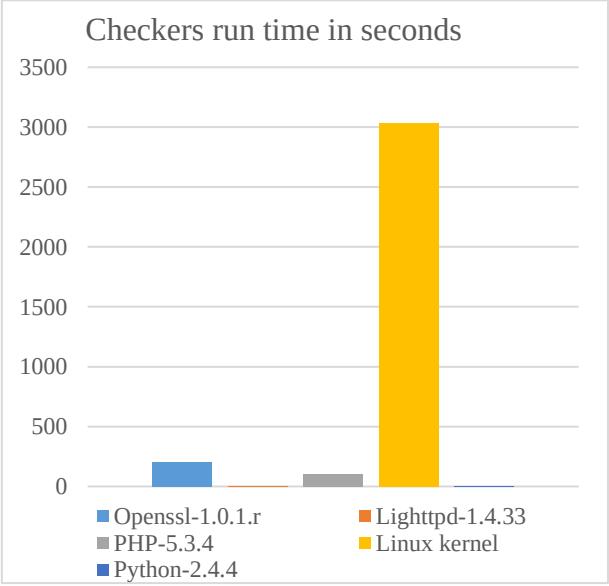
3. RESULTS

We have performed analysis both on synthetic examples and real world projects. In Picture 4 you can find simple "use after free" example, which is detected by the instrument.

```
#include <iostream>
int f(int *p) {
  return (*p)++;
}
int main() {
  int *b = new int[10];
  f(b);
  delete[] b;
  int p = *b;
  return 0;
}
```

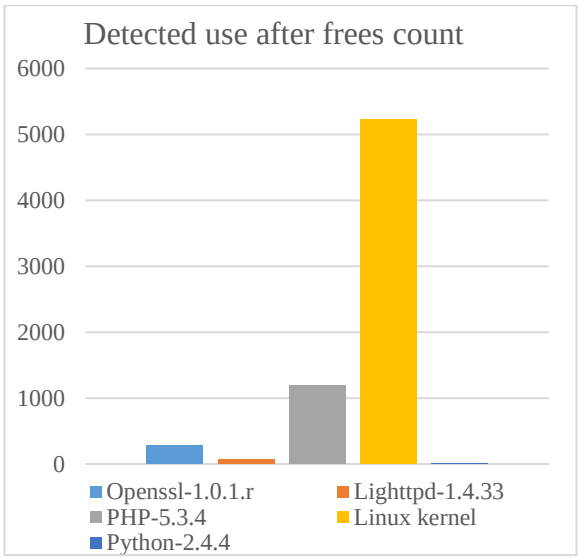
Picture 4.

Picture 5 demonstrates the checkers analysis time for projects *openssl*, *libhttpd*, *php*, *linux kernel* and *python*. The kernel of *linux* was analyzed less than in one hour. The analysis of *libhttpd* took only three seconds.

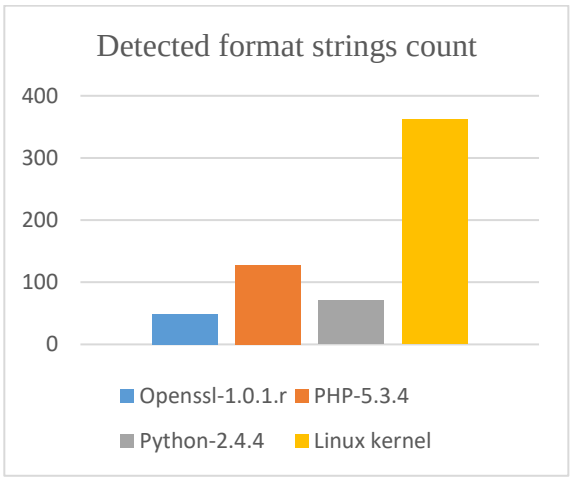


Picture 5.

Picture 6 and Picture 7 show the number of alerts for "use after free" and "format string" checkers. For *linux kernel* the instrument detected about 5200 potential use after frees. For *openssl* only 276 potential defects are found.



Picture 6.



Picture 7.

Manual analysis shows that two of the detected defects in *openssl* are already known CVE-2016-0705 and CVE-2016-0799. Alarms analysis shows that currently we have high rate of false positive (about 80%). It is because of some inaccuracies in SDG and not fully tuned checkers. We look forward for farther improvements.

4. FUTURE WORK

We have two major priorities for proposed framework improvement. The first one is the more accurate SDG generation from binary code. For that purpose, we plan to extend calling convention functionality to support custom calling conventions and perform alias analysis. It would lead to more precise SDG generation and the work of checkers will be more accurate.

The second one is reduction of false positive. We will add number of filters for that purpose. For example, some filters will check corresponding source and binary code locations.

Two improvements we plan to add buffer overflow and integer overflow checkers on SDG.

REFERENCES

[1]. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. Trans. on Prog. Lang. and Syst., 12(1):26–60, January 1990.

[2]. www.hex-rays.com/products/ida

[3]. https://www.zynamics.com/binnavi.html

[4]. R. M. Stallman and GCC-Developer Community, *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA: CreateSpace, 2009.

[5]. K. Chen and D. Wagner, “Large-scale analysis of format string vulnerabilities in debian linux,” in *PLAS*, 2007, pp. 75–84.

[6]. W. L. 0020 and T. cker Chiueh, “Automated format string attack prevention for win32/x86 binaries,” in *ACSAC*, 2007, pp. 398–409.

[7]. P. Kohli and B. Bruhadeshwar, “Formatshield: A binary rewriting defense against format string attacks,” in *ACISP*, 2008, pp. 376–390.

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

[9].www.mathworks.com/training-schedule/polyspace-code-prover-for-cc-code-verification.html.

[10]. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c—a software analysis perspective. In: SEFM, pp. 233–247 (2012).

[11]. F. Yamaguchi, A. Maier, H. Gascon, K. Rieck, "Automatic Inference of Search Patterns for Taint-style Vulnerabilities", Proceedings of the 2015 IEEE Symposium on Security and Privacy, pp. 797-812, 2015-July.

[12]. Grammatech, “Codesurfer,” www.grammatech.com

[13]. B. Scholz, C. Zhang, and C. Cifuentes, “User-input dependence analysis via graph reachability,” in IEEE Int. Workshop on Source Code Analysis and Manipulation, Los Alamitos, CA, USA, 2008, pp. 25–34.

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

[15]. V. Chipounov and G. Candea. Enabling sophisticated analyses of x86 binaries with RevGen. In Workshop on Dependable Systems and Networks, 2011

[16]. https://github.com/trailofbits/mcsema