Reachability Confirmation of Statically Detected Defects Using Dynamic Analysis

Alexander Gerasimov
Institute for System Programming of the Russian Academy of Sciences
Moscow, Russian Federation
e-mail: agerasimov@ispras.ru

Leonid Kruglov
Institute for System Programming of the Russian Academy of Sciences,
Lomonosov Moscow State University
Moscow, Russian Federation
e-mail: kruglov@ispras.ru

ABSTRACT

Static and dynamic analysis of programs are well-known approaches to the problem of automatic program behaviour analysis. Both methods have advantages and limitations. Static analysis has lack of precision for the sake of scalability. On the other hand, dynamic analysis has 100% precision while reaching the defect point, but suffers from scalability issues. This paper describes an approach to confirmation of reachability of source-sink defects that were found by static analysis with help of dynamic analysis. The combination of methods allows to circumvent their limitations and multiply their advantages. Preliminary experiments on several open source projects show that real true positive defects can be confirmed to be reachable using the approach and some false positives can be proved.

Keywords

Static analysis, dynamic analysis, source-sink defects, defect reachability confirmation.

1. INTRODUCTION

Nowadays, software experiences constant complexity growth and the problem of automatic discovery and reproduction of program defects becomes very actual. There are at least two alternative approaches to the problem of automatic software analysis: static analysis and dynamic analysis.

Static analysis is performed without program execution. Static analysis tools generally analyze source code, binary code is analyzed less often. Static approach is usually based on building and analyzing semantic model of a program. In practice, such models are quite abstract and may exclude some details of program behaviour for the sake of analysis performance, hence static analysis is incomplete and false positive error warnings are present. On the other hand, main advantage of static analysis is scalability, which makes it possible to analyze programs by relatively small fragments and achieve high code coverage during analysis.

On the contrary, dynamic approach is applied to the program under analysis either during the process of program execution (online analysis), or after the program stops (offline analysis, post-mortem analysis). Code instrumentation is often performed to collect information needed for dynamic analysis without changing program behaviour greatly. Under the assumption of deterministic program behaviour, when the same execution path is followed during program execution on the same set of input data, it is possible to apply the technique of dynamic symbolic execution for calculation of new inputs which will guide the program through alternative path [1]. The problem of dynamic approach is that defects which reside on unexplored paths can never be found. Another well-known problem is exponential growth of path number which prevents the usage of dynamic symbolic execution in practice [2].

Consider a group of program defects (source-sink defects) that are discovered by static analysis of execution paths from the point of initialization of a possible error (the source) to its trigger point (the sink). Critical errors like null pointer dereference, division by zero, unhandled exceptions and vulnerabilities like precision loss, improper use of format strings, buffer overflow are examples of source-sink defects.

After such defect is detected and reported by a static analysis tool, the following questions arise:

1. is the source of the defect reachable?
2. are source-sink path conditions satisfiable?
3. can the defect point reachability be confirmed?

In the general case, it is impossible to prove reachability of the source of the defect. Existence of branch targets computed at runtime is one of corresponding limitations of static analysis. The problem of data under-tainting is the reason for existence of missing execution paths during iterative dynamic analysis [3, 4]. Feasibility of source-sink path conditions can be checked by applying a SMT solver (like STP [5]) to the formulas describing the path. So, it is possible to prove a defect to be false positive only if corresponding source-sink path conditions are infeasible.

Thus, given a warning message with information about the point in a program with the source of a defect and the source-sink path, the proposed dynamic analysis approach to confirmation of defect reachability is the following:

- reach the source of the defect using iterative dynamic analysis;
- for each path through the source of the defect try to reproduce the source-sink path;
- report the defect point to be reachable if the source sink path is successfully reproduced.

The paper is organized as follows: in Section 2 the proposed approach is described, Section 3 covers experimental results, overview of related work can be found in Section 4, Section 5 contains the conclusion and directions of future work.

2. PROPOSED APPROACH

Required input information for the proposed approach is a warning for a potential defect that is generated by a static analysis tool and executable program. The warning contains detailed information about source instruction of defect, trace from source to sink of the defect in terms of control flow transitions on conditional operators. Code optimizations are a serious obstacle for the task of establishing correspondence between source code trace and binary code instructions and that is why the executable program is assumed to be built without optimizations using the same version of source code that was analyzed by a static analysis tool.

Our approach is based on the idea of directed dynamic analysis described in our previous work [6] which is iteratively applied to the defect point reachability task.

2.1. Minimal distance path selection

First task we have to complete for defect reproduction is reaching the source point of potential defect. The following heuristics is proposed: on every step of iterative analysis a path is selected if it is the closest available path on control-flow graph to the source of potential defect.

To collect all required information that is used during minimal distance heuristics calculation it is sufficient to build an incomplete call graph: only the nodes from which it is possible to reach the source of the defect need to be included in the graph. The graph is generated backwards starting from the function containing the source of the error. First, all call points of the function are found. Then for each call point its enclosing function is determined and new call points are found. The process continues until a function with no call points is reached: program entry point, thread function, function called via pointer or unreachable function. If recursion is detected, corresponding functions should be skipped as they have already been processed. Next, control flow graph of functions is processed additionally to call graph: for each node, an incomplete control flow graph is built which contains basic blocks from the paths starting at node entry and finishing at call points of other nodes of the call graph. Once such extended graph is built, distances from defect source to other nodes of the graph can be efficiently calculated.

Thus, the process of building defect source call graph has the following steps:

- the function with the source of the defect is determined and its call graph is built;
- incomplete control flow graph is built for each node of the call graph;
- the graphs are merged into extended control flow graph;
- distances from the source of the defect to all nodes of the graph are calculated.

During the process of minimal distance heuristics calculation, it is sufficient to process only the basic blocks that are present in the defect source call graph because other blocks do not affect defect source reachability and can therefore be ignored. Suppose that some program execution path contains basic blocks A, B and C and the defect source point is reachable from them. Then the value of heuristics is defined by the following formula:

```
e = min(distance(A, T), distance(B, T),
             distance(C, T))
```

`T` is the node that corresponds to the source of the defect, `distance(X, T)` is the distance from node `X` to node `T` in the graph.

So, the result of heuristics calculation for some execution path is the distance from the closest to the defect source node among all nodes in the path. Execution path is considered to be of higher priority if the corresponding heuristics value is the least. Heuristics value is equal to 0 if and only if the source of the error has been reached.

2.2. Source-sink path transformation

After the source of potential defect is reached, it is proposed to reproduce the source-sink path using directed dynamic analysis: execution of the program is altered to match the source-sink path. As the source-sink path in the warning message is given in terms of source code, it needs to be transformed into a path in terms of binary code so that it can be used during dynamic analysis. The transformation approach was designed for C source code and unoptimized binary code that is generated using GCC [7] and was tested on GCC versions 4.6.3 and 4.9.2 for x86-64. Currently, transformation for `if`-statements is supported. Loops, ternary operator and operator `switch` are out of scope of current work and is included in future research. Under the assumptions stated above, the order of conditional jumps in binary code matches the order of the parts of the `if`-statement condition (conjuncts and disjuncts).

Conditional jump in the binary code is either a jump to a basic block that corresponds to then-branch in source code or a jump to else-branch basic block. Code generation rules for complex conditions with conjunction, disjunction and negation are given below:

1. Conjunction "`A && B`"

- jump to then-branch: else-branch for `A` and then-branch for `B`;
- jump to else-branch: else-branch for `A` and else-branch for `B`.

2. Disjunction "`A || B`"

- jump to then-branch: then-branch for `A` and then-branch for `B`;
- jump to else-branch: then-branch for `A` and else-branch for `B`.

3. Negation "`!A`"

- jump to then-branch: else-branch for `A`;
- jump to else-branch: then-branch for `A`.

Jump to then-branch or else-branch during code generation is chosen depending on the type of operations inside the branches and configuration of parent `if`-statements. The following property was detected:

- jump to then-branch is generated if corresponding then-branch in the source code contains a single jump operation: `goto`, `break` or `continue`;
- however, jump to else-branch is generated in case `if`-statement has empty else-branch and is contained inside then-branch of parent `if`-statement (not necessarily immediate parent);
- jump to else-branch is generated in all other cases.

Thus, transformation of single source-sink path in terms of source code can result in several paths in terms of binary code and there may exist several different sets of input data that correspond to single warning message.

2.3. Source-sink path reproduction

After the source of possible defect is reached by iterative analysis, directed dynamic analysis is performed to reproduce source sink path. It allows to reduce the time required to reach the point of realization of the error compared to iterative traversal of program branch tree. Path conditions are collected during directed analysis to check feasibility of resulting execution path and to generate input data for its reproduction, if possible.

Alternation of tainted branch conditions leads to execution of a new path and, what is most important, successful generation of corresponding input data set. However, in case of untainted branch conditions, it is useless to track their alternation because they do not affect input data and it is therefore impossible to generate corresponding inputs to reproduce the path. If source-sink path contains a branch with untainted condition, then the result of directed analysis can be incorrect and three situations are possible:

- branch condition is not dependent on input data and needs to be altered;
- branch condition is implicitly dependent on input data and needs not to be altered;
- branch condition is implicitly dependent on input data and needs to be altered.

In the first case, it can be concluded that provided source-sink path is not feasible. Second case corresponds to the situation when branch condition in source-sink path matches actual condition during program execution under directed analysis and the resulting path is hence feasible. In the third case, there exist two alternatives:

- altered condition does not match the previously executed path and the result of directed analysis is incorrect;
- defect realization actually does not depend on the altered branch condition which was added to the source-sink path by static analysis tool by mistake, so the result of directed analysis is correct.

Therefore, manual inspection of the defect and generated input data is needed if untainted branch condition is changed during directed dynamic analysis.

The result of performed directed analysis is a system of formulas that describes the constraints on the data which the program operates with. If branch condition is altered, a new formula

```
  tainted_condition_var != old_condition
```

needs to be added to the system (`tainted_condition_var` is the tainted variable holding the condition, `old_condition` is its value prior to alternation). If such system of formulas is not satisfiable, the resulting path is not feasible and iterative search of another path to the source of the defect should be continued.

3. EXPERIMENTAL RESULTS

The proposed approach was implemented on top of Avalanche [2] iterative dynamic analysis engine. Minimal distance heuristics was added to the instrument and tainted data propagation module was modified to implement directed dynamic analysis. Defect source call graph generation module was implemented as a parser of disassembled by objdump [8] binary code. Source sink path transformation was implemented as an abstract syntax tree (AST) processor, ASTs for source files are expected to be provided by static analysis tool.

Reference static analysis tool which was used to evaluate the approach has detected source-sink defects of the following types: null pointer dereference (NPD), buffer overflow (BOF), resource leak (LEAK), usage of tainted data without prior validation (TAINT), usage of uninitialized values (UNINIT).

Table 1 contains results of analysis for eight open source projects included into packages of Debian Linux. Total 304 defects were analyzed, 110 defects were classified as false positives (columns F) during manual code inspection and 194 – as true positives (columns T). The proposed approach was applied to each defect detected by static analysis tool. Dynamic analysis was launched with 30-minute time limit. Columns C of *Table 1* contain information about the number of defects for which source-sink path was successfully followed which means that corresponding defect execution path is considered to be confirmed and defect point is considered to be reachable with all path conditions reported by static analysis tool.

Total 27 source-sink defects paths were successfully reproduced. However, both null pointer dereference defects were actually classified as false positives during code inspection. Such successful source-sink path reproduction and hence false confirmation of the defect execution paths is explained by current limitations of the approach: realization of these defects additionally requires satisfiability of constraint

```
             variable == NULL
```

which corresponds to the following code at the source of the defect:

```
   variable = someFunction(arguments);
```

In current implementation of the approach the information about function calls is ignored which leads to false confirmation of defect execution paths.

Other 25 defects for which source-sink paths were successfully reproduced were classified as true positives during code inspection. Thus, the defects can be considered as successfully reproduced. 9 defects required manual inspection as untainted branch condition was changed during source-sink path reproduction and they were classified as false positives.

Project	NPD			BOF			LEAK			TAINT			UNINIT		
	T	F	C	T	F	C	T	F	C	T	F	C	T	F	C
a2ps	33	65	2*	5	2	0	22	2	11	7	0	2	0	0	0
bbe	2	0	0	10	0	0	0	1	0	0	0	0	0	0	0
bc	0	0	0	0	0	0	6	1	4	1	0	1	0	0	0
byacc	0	0	0	0	10	0	1	0	0	0	0	0	2	0	0
bzip	0	0	0	1	0	0	14	0	0	9	0	0	1	0	0
ccrypt	0	0	0	0	0	0	15	7	2	1	0	0	0	0	0
chrpath	0	0	0	0	0	0	9	0	0	10	0	3	0	0	0
cpio	22	19	0	0	0	0	18	2	2	4	0	0	1	1	0
total	57	84	2*	16	12	0	85	13	19	32	0	6	4	1	0

Table 1. Defect confirmation statistics

4. RELATED WORK

There exist several tools for defect discovery and confirmation. They differ in programming language of analyzed program, type of found and confirmed defects, level of supported static analysis (source vs binary code), applied dynamic analysis approach and mechanism of input data generation for defect reproduction.

JCHARMING [9] and STAR [4] are analysis instruments targeted at reproduction of unhandled exception errors in Java applications. First, information about the point where exception is thrown is extracted from call stack trace. Next, JCHARMING generates a sequence of operations for error reproduction using the mechanism of model checking [10]. STAR uses backward symbolic execution for creation of path conditions from error sink to entry point of analyzed program which are later used to generate a sequence of operations to reproduce the defect.

ConDroid [11] and IntelliDroid [12] are tools for analysis of Android applications that use static analysis techniques to identify call points of vulnerable programming interfaces and to build paths leading to the points. Dynamic symbolic execution is performed to generate sequences of system and user events to reproduce the paths that are created by static analysis.

The goal of DyTa [13] tool is to find and reproduce defects of C# programs. Supported defects are user-specified contract violation, null pointer dereference and division by zero. At the first step, static analysis is used to find possible defects and generate paths to them. This information is used at the next step by iterative dynamic analysis to generate input data to reproduce the errors.

AEG [14] instrument statically analyzes LLVM byte-code [15] to detect format string and buffer overflow vulnerabilities. Dynamic symbolic execution is utilized to generate inputs which correspond to execution paths to the sink of the vulnerabilities. Finally, these inputs are used during dynamic analysis which is performed to collect low level information to generate exploits.

An approach to buffer overflow defect discovery and reproduction is presented in paper [16]. Defects are detected using static analysis of visibly pushdown automaton (VPA) [17] which represents control flow graph of analyzed program. The automaton is later used during dynamic symbolic execution to generate input data to reach error sinks. The imprecision of VPA static analysis may result in false positive error warnings.

Approach described in paper [18] classifies memory leak warning messages into four classes: “error confirmed”, “error unlikely”, “error possible” and “ineffective code”. Iterative dynamic analysis is used to generate inputs on which error sinks can be reached. Control flow graph based error sink reachability analysis is used to improve iterative path-guided concolic testing.

Paper [19] presents an approach to reproduction of buffer overflow defects in C programs. Static analysis is done to build partial paths to a point in the program with potential vulnerability. Such paths contain information about tainted data only. Next, single path is selected which contains maximal number of unique nodes. Genetic algorithm is applied during dynamic analysis to generate input data on which selected path is most precisely reproduced. Statically generated paths to vulnerabilities may be infeasible.

5. CONCLUSION AND FUTURE WORK

If source to sink path can be proved as infeasible it will show that the reported defect is false positive. Current approach allows to confirm reachability of defect point. If source to sink path is successfully followed then the defect is considered to be reachable. On the other hand, our experiments showed that it is not enough to confirm defect reachability itself. Main directions of future work are elimination of known limitations in order to remove existing drawbacks such as under-taintedness and extend the set of defects that can be confirmed.

REFERENCES

[1] J.C. King. Symbolic execution and program testing. *Communications of the ACM*. Vol. 19, Issue 7, pp. 385-394, 1976

[2] I.K. Isaev, D.V. Sidorov. The Use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. *Programming and Computer Software*. Vol. 36, Issue 4, pp. 225-236, 2010

[3] E.J. Schwartz, T. Avgerinos, D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. pp. 317-331, 2010

[4] N. Chen, S. Kim. STAR: stack trace based automatic crash reproduction via symbolic execution. *IEEE Transactions on Software Engineering*. Vol. 41, Issue 2, pp. 198-220, 2015

[5] V. Ganesh, D.L. Dill. A decision procedure for bitvectors and arrays. *Computer Aided Verification*. pp. 519–531, 2007

[6] A.Y. Gerasimov, L.V. Kruglov. Input data generation for reaching specific function in program. *Trudy ISP RAN/Proc. ISP RAS*. Vol. 28, Issue 5, pp. 159-174, 2016 (in Russian)

[7] R. Stallman et al. Using the GNU Compiler Collection. *Free Software Foundation*. 2004

[8] GNU Binutils [HTML] (http://www.gnu.org/software/binutils/)

[9] M. Nayrolles et al. JCHARMING: A bug reproduction approach using crash traces and directed model checking. *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. pp. 101-110, 2015

[10] C. Baier, J.P. Katoen, K.G. Larsen. Principles of model checking. *MIT press*, 2008

[11] J. Schütte, R. Fedler, D. Titze. Condroid: Targeted dynamic analysis of android applications. *IEEE 29th International Conference on Advanced Information Networking and Applications*. pp. 571-578, 2015

[12] M.Y. Wong, D. Lie. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*. 2016

[13] X. Ge et al. DyTa: dynamic symbolic execution guided with static verification results. *Proceedings of the 33rd International Conference on Software Engineering*. pp. 992-994, 2011

[14] T. Avgerinos et al. Automatic exploit generation. *Communications of the ACM*. Vol. 57, Issue 2, pp. 74-84, 2014

[15] C. Lattner, V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. pp. 75-86, 2004

[16] D. Babić et al. Statically-directed dynamic automated test generation. *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. pp. 12-22, 2011

[17] R. Alur, P. Madhusudan. Adding nesting structure to words. *Journal of the ACM (JACM)*. Vol. 56, Issue 3, Article 16, 2009

[18] M. Li et al. Dynamically validating static memory leak warnings. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. pp. 112-122, 2013

[19] S. Rawat et al. Combining Static and Dynamic Analysis for Vulnerability Detection. *arXiv preprint arXiv:1305.3883*, 2013