Detection of Incorrect Pointer Dereferences for C/C++ Programs using Static Code Analysis and Logical Inference

Tatiana Vert
St. Petersburg State
Polytechnical University
St. Petersburg, Russia
Email: werttanya@gmail.com

Tatiana Krikun
St. Petersburg State
Polytechnical University
St. Petersburg, Russia
Email: krikunts@gmail.com

Mikhail Glukhikh
Clausthal University
of Technology
Clausthal, Germany
Email: mikhail.glukhikh@gmail.com

***Abstract*—This article considers a method for an increase of static code analysis precision. The method extends classic code analysis algorithm with dependency analysis. For this purpose, during abstract interpretation information about statically known values should be extracted as well as dependencies between unknown values. Dependencies can be represented with first-order logic predicates. Such a method allows using of external logical inference tools to prove truth or falsehood of branch conditions and of error occurence conditions. The main focus is oriented to pointer analysis logic and incorrect dereference detection rules. A prototype is implemented and results of efficiency evaluation are provided. The prototype uses Microsoft Z3 Solver as a logical inference tool. A significant precision increase is shown, ways for performance boosting are suggested.**

Keywords*—C source code error detection, static code analysis, logical inference.

I. Introduction

Software quality assurance is a key problem in the modern world. It's well known [1] that even good-tested software could contain up to 0.7 errors in a thousand lines of code. Incorrect pointer operations belong to the most common errors.

The primary software quality assurance method is testing. Despite of testing advantages we should mention, that it gives no guarantee for every error detection. That's why sometimes testing cannot be the only quality assurance method. Such a guarantee could be provided by static methods, code analysis methods are among them [2][3].

Error detection methods have three key characteristics, which are soundness, precision and performance. *Soundness* can be calculated as a ratio of true detected errors among all errors in a program. *Precision* is a ratio of true detected errors among all detected errors. These three characteristics are in contradiction, and each error detection tool must choose a compromise. For example, industrial static analyzers [4] are usually oriented on high precision and performance, but have no guarantee of soundness. Research analyzers often try to detect every error of particular class, thus providing 100% soundness. However, that's why they have low precision and performance, and not so much in use in industry [3].

Abstract interpretation is a well-known code analysis algorithm [5]. This algorithm operates like a common interpretation, but analyses every execution path without violating soundness of analysis. Abstract interpretation uses so-called abstract semantic domains for variable values representation. Examples of such domains are the set of all integer intervals or the set of all pointers to data memory. Abstract interpretation allows to analyze paths separately thus providing high precision but very low performance. Usually it is inacceptable, so as an alternative, joint path analysis could be used which assumes joining variable values at flow conjunction points. That leads to low precision but acceptable performance.

A possible method to increase precision while using joint path analysis is extending classic analysis algorithms by dependency analysis [6]. A dependency is any relation between two or more variable values, mathematically it can be represented as a predicate [7]. Dependency analysis allows to partially compensate precision loss which occurs because of joining paths during code analysis. Dependency analysis can be implemented using existing logical inference methods and tools, e.g. Horn clauses and logic programming tools [8], first and higher order logic [7], SMT solvers [9]. Such a possibility allows also to simplify other algorithms in use, e.g. to use exact value analysis instead of interval analysis. In [10] Patric Cousot considers a mathematical model of combining abstract interpretation with a logical inference. As we know, at this moment this model is not yet implemented as a static analysis tool.

The key idea of this article is to use a combination of code analysis and a logical inference for source code error detection. Incorrect pointer operations were chosen as the primary error group, but a proposed approach could be extended for other error detection. Chapter 2 reviews related works. Chapter 3 considers the proposed approach. Chapter 4 is dedicated to an implementation of the approach and further efficiency evaluation. Chapter 5 concludes the article and outlines directions for a future advancement.

II. Related Works

Nowadays, there is a lot of either commercial or research static code analysis tools [11]. The commercial leader is now de-facto Coverity SAVE platform [12]. This tool can analyze programs of arbitrary size and provides high precision, which is from 80% to 90% according to description from its authors.

978-0-9860-7731-9/14
DOI 10.1109/TMPA.2013.12

The tool has no guarantee of soundness, but is able to detect a vast number of errors, like 7 errors in 10 thousand lines of code, in real open-source projects [4], [1]. Its approach is based on extraction of different rules [13] from a source code. This rules describe how different program objects can be used. Either exact rules like “P pointer is not NULL” or statistic rules like “M mutex possibly protects V variable” exist. After extraction this rules are checked for consistency to detect errors in a program.

The research tool Astree [14] is developed under Patric Cousot leadership. The tool searches for run-time errors in C programs and was used to analyze different industrial medium-size projects, around 100000 lines of code. According to [15], the tool uses well-known abstract semantic domains, like integer intervals or pointer sets, as well as different domains for dependency tracking, like an octagon domain or a linear expressions domain. After source code parsing Astree begins with light analysis to simplify the main task, e.g. detection of a dead code. Then it executes the main analysis procedure from the top to the bottom, so beginning from a program entry point and finishing with the most simple functions.

The research tool Saturn [16] is based on bottom-up analysis [17]. Information received during simple function analysis is used for their callers analysis, so each function is analysed only once. The same principle can be used for loop analysis. A constraint inference is used in different execution points, and after that SAT-solver is executed to determine satisfiability. The logic programming language Calypso is used to describe analysis rules. The tool can be used for large program analysis but according to [18] it has pretty low precision.

The PREfix tool also utilizes bottom-up analysis [19]. Well-known data flow analysis algorithms and pattern matches algorithms are used, together with some other algorighms. Particularly an idea of program state description as a set of exact values extended by a set of predicates is used [20].

The PC-Lint tool for MS Windows and the FlexeLint tool for Unix/Linux/MacOs/Solaris from Gimple Software [21] detect errors in C/C++ program code. Tools execute parsing, data flow analysis and control flow analysis, also provide interprocedural analysis. Special checking methods are utilized such as value analysis, additional strong type checking, user-defined semantic checking of function arguments and return values.

At last, the Aegis [22] research tool analyzes programs from the top to the bottom. The tool utilizes pointer analysis, interval analysis, resource analysis and uses simple dependency analysis [6] to increase precision.

III. Description of the developed approach

The proposed approach uses logical inference tools for proving different assertions during code analysis. Particularly, assertions on the presence or absence of defects in statements of a program are proved.

A. Architecture

The first step of code analysis is construction of a source code model for a program. The most convenient model which can be used for code analysis is a *control flow graph (CFG)*. Nodes of this graph correspond to executable program statements.

The next step is an application of code analysis algorithms in order to determine a state of a program at all execution points. Algorithms determine all possible values of program variables using known information about language semantics.

At the last step, which is performed in parallel with the previous one, defect detection algorithms are applied. They use program states at different execution points for detection and localization of available errors.

To use logical inference tools during abstract interpretation, it is necessary to provide input data in a suitable form. One of the options for such a representation is to save a program state as a set of *predicates* and then use this set for inferencing.

The general scheme of the analyzer, reflecting the structure of the proposed approach, is represented in Fig.1.

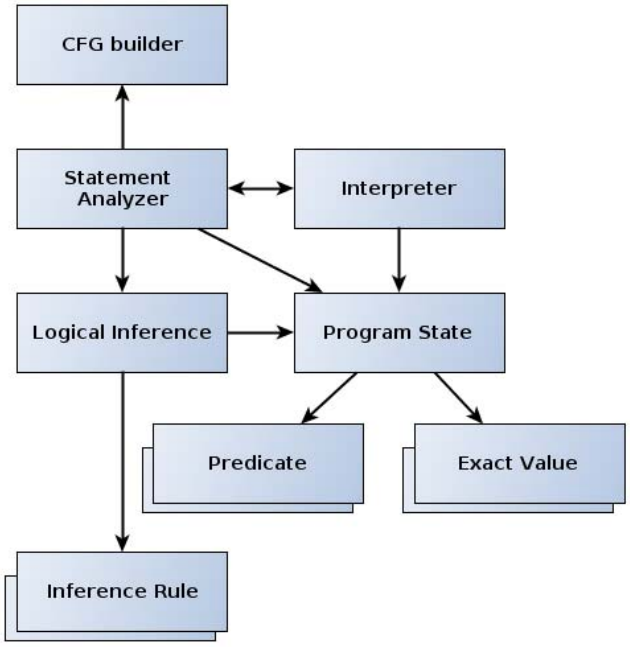

Figure 1. The structure of the proposed approach

The approach assumes deducing and saving exact values of variables at a current moment of interpretation. If it is impossible to determine an exact value of a variable statically, then different relations with other variables can be stored instead. Thus, dependencies between variables are added to the exact values.

An inference tool is responsible for interaction with an external prover. It is assumed to prove the truth of certain predicates and to simplify a set of predicates to reduce its size.

The analyzer searches for defects in specific statements using a current state of a program and inference results.

B. Predicate extraction

Information about exact values of variables and relations between them is extracted during analysis of program statements. These values and relations can be represented as logical *predicates* [7][8]. In the simplest case (e.g., interpretation of assignment statements), interpretation of a statement creates a new predicate, which describes the statement semantics exactly.

In our approach, a *predicate* is represented as a formula $p(v_1, v_2, ..., v_n)$, where p is a functional symbol and $v_i, i = 1 \dots n$ are *predicate variables*. The value of a predicate variable usually corresponds to the value of a program variable. There are examples of these predicates:

- arithmetic: $sum(a, b, c)$ or $a = b + c$, $diff(a, b, c)$ or $a = b - c$;
- comparison: $equals(a, b)$ or $a = b$, $greater(a, b)$ or $a > b$, $less\_equals(a, b)$ or $a \leq b$.

More complex predicates can contain other predicates as their variables. These predicates include:

- $oneof(p_1, p_2)$: at least one of the predicates p_1, p_2 is true,
- $oppos(p)$: the predicate p is false,
- $equiv(p_1, p_2)$: the predicates p_1 and p_2 are simultaneously true or simultaneously false.

These complex predicates can also be described in terms of the first-order logic, without using predicates as arguments, therefore their addition does not increase the order of logic. The above definition of complex predicates is chosen for ease of implementation.

One program variable can be described by more than one *predicate variables*. These predicate variables describe its value in different execution points. This requirement is important, because the first-order logic does not function correctly if values of involved variables can be changed. Therefore it was decided to represent variables on the basis of the static single assignment. Thus, in each direct assignment a variable in the left part of a statement is used in a new context, and therefore a new predicate variable is created. For example, interpretation of the sequence `a=b+c; d=a; a+=e` creates the set of predicates $sum(a_1, b_1, c_1)$, $equals(d_1, a_1)$, $sum(a_2, a_1, e_1)$.

Description of logical operations is specific for the C language, logical values are represented as integers because of the language semantics. The zero corresponds to the falsehood, and nonzero to the truth. Corresponding predicates are $zero(v)$ and $nonzero(v)$, the first of them describes zero or the falsehood, the second describes nonzero or the truth. Predicates $conj(a, b, c)$ (`a = b && c`), $disj(a, b, c)$ (`a = b || c`) describe the C logical operations, and predicate $equiv(p_1, p_2)$ can be used for creating relations between logical and arithmetic variables. For example, interpretation of the sequence `g = (a>0); h = (b>=a); f = g && h` creates the set of predicates $equiv(nonzero(g_1), greater(a_1, 0))$, $equiv(nonzero(h_1), greater\_equals(b_1, a_1))$, $conj(f_1, g_1, h_1)$.

During interpretation of a conditional statement `if(cond)`, where `cond` is an expression, two output states are created from one input state. If `cond` does not have an exact value, then the logical inference tool is used. This tool checks if the condition is true or false using a current set of predicates. If some branch is dead, so the condition is never true or never false, then further analysis for this branch is not performed. After condition checking, a predicate corresponding to the truth of `cond` is added to the state of the true branch. Similarly a predicate corresponding to the falsehood of `cond` is added to the state of the false branch. For example, if `cond` is a variable, predicates $nonzero(cond)$ and $zero(cond)$ are used respectively.

During interpretation of a control flow conjunction point, or so-called ϕ-function, input program states are merged to reduce the number of analyzed paths. Unlike other statements a ϕ-function has two or more input statements and two or more predicate states at inputs respectively. The basic rules of the state merging are:

- if a predicate variable v_i has different values x_1, x_2 in input states, then disjunction of its possible values x_j is represented as a predicate $oneof(equals(v_i, x_1), equals(v_i, x_2))$;
- if a program variable v corresponds to different predicate variables v_1 and v_2 in the input states, then the new predicate variable v_3 is created and the predicate $oneof(equals(v_3, v_1), equals(v_3, v_2))$ is added to the output state;
- all predicates from the input states are added to the output state.

To describe the work with pointers and complex variables of the C language, like structures or arrays, we use an additional set of predicates:

- a complex object element $arr(a, v, s)$: an array or a structure a contains a value v with an offset of s bytes, or `a[s]=v` in case of a byte array;
- size of a complex object $sizeof(a, s)$: an array or a structure a has a size of s bytes, or `sizeof(a)=s`;
- pointer to an object $ptr(a, p, s)$: a pointer p points to an array (structure, variable) a with an offset of s bytes, or `p=(void*)(&a)+s`;
- a pointer dereference $deref(p, v)$: a variable value v equals to a value pointed by a pointer p, or `v=*p`;
- pointer correctness $correct\_ptr(p)$, $incorrect\_ptr(p)$: a pointer p has a correct value, thus points to a real existing variable, or has an incorrect value, thus it is the null pointer or a pointer to an inexistent variable.

Sometimes it is important to consider a relation between a pointer value and a conditional expression during branch merging. In general, if a variable has different values in different branches of a conditional statement, a predicate which describes its value should be associated with the conditional expression for increasing precision. Let's consider this example:

```
if(size > 0)
        q = malloc(size);
else
        q = 0;
```

After interpretation of these statements, the state in the true branch contains predicates $greater(size_1, 0)$, $sizeof(dyn, size_1)$ and $ptr(dyn, q_1, 0)$, where dyn is a dynamic array. The state of the false branch contains predicates $less\_equals(size_1, 0)$ and $equals(q_2, 0)$.

During merging these two states the relation between a value of the variable `size`, a value of the pointer `q` and the fact of a dynamic array existence is important. In further analysis of a free memory statement, analysis should know that the dynamic array exists only when `size > 0`, or when `q` is not the null pointer. So the output state should include this dependency, otherwise precision is lost.

So in the different branches of conditional statement the pointer `q` corresponds to the different predicate variables q_1, q_2 at the moment of states merging. According to the above rules the new variable q_3 is created and the predicate $oneof(equals(q_3, q_1), equals(q_3, q_2))$ is added. To take into account the dependency between the variables `size` and `q` the predicate $equiv(less\_equals(size_1, 0), equals(q_3, 0))$ should also be added to the state.

C. Pointer analysis rules

Logical inference is performed using a set of axioms, and all conclusions are based on this set. The basis of all logical inference tools is a basic set of axioms for common theories, like the theory of integers. Since the theory of pointers is non-standard for logical inference tools, it is necessary to extend a basic set of axioms. Thus, to make pointer analysis by using logical inference tools, the following rules should be introduced.

1) Pointer correctness:

$$\frac{ptr(t, p, s), sizeof(t, v), greater\_equals(s, 0), less(s, v)}{correct\_ptr(p)} \tag{1}$$

a pointer p is correct, if its offset s for a target t is non-negative and less than the size v of t;

2) Dereference of a pointer to a simple variable:

$$\frac{ptr(t, p, 0), deref(p, v)}{equals(t, v)} \tag{2}$$

if p points to t with the offset 0, then a value v, which is pointed by p, equals to t;

3) Dereference of a pointer to an element of a complex object:

$$\frac{ptr(a, p, s), arr(a, t, s), deref(p, v)}{equals(t, v)} \tag{3}$$

if p points to a with an offset s and t is a part of a with the offset s, then a value v, which is pointed by p, equals to t.

4) Sum of pointer and integer constant:

$$\frac{ptr(a, p, s), sum(t, s, b), sum(q, p, b)}{ptr(a, q, t)} \tag{4}$$

if p points to a with an offset s, t is the sum of s and b, a pointer q equals to the sum of p and b, then q points to a with the offset t.

The following rules are used to detect defects associated with pointer operations.

D. Defect detection rules

In this article, the following types of defects are under consideration:

- incorrect pointer dereference;
- buffer overflow and array out-of-bounds.

An incorrect pointer dereference is a defect associated with a dereference of an incorrect pointer or the null pointer. The result of a dereference of an incorrect pointer is an error. An example of an incorrect pointer is a pointer to the released memory. Defect detection procedure depends on the pointer's value during interpretation of indirect read/write statements. If a pointer has an exact value of `null` or an incorrect value, then a defect is detected. If a pointer does not have an exact value, the presence of a defect in a statement can be confirmed by proving *satisfiability* of certain predicates using a set of existing predicates as a base. These predicates to prove are:

- null pointer: $equals(ptr, 0)$;
- incorrect pointer: $oppos(correct\_ptr(ptr))$.

To demonstrate the absence of a defect, it is necessary to prove the opposite predicates:

- non-null pointer: $nonzero(ptr)$;
- correct pointer: $correct\_ptr(ptr)$.

Array out-of-bounds detection is performed during interpretation of a dereference operation `*ptr` or an access by an index operation `ptr[i]`. For each pointer value `(obj, shift)` we should check that the offset $shift \geq 0$ and $shift < sizeof(obj)$. If this condition is provable, then there is no defect, otherwise a defect is detected.

IV. Efficiency Evaluation

For efficiency evaluation a prototype tool was implemented. In the tool, control flow graph is built using gcc plugin. Aegis platform [22] was used as a tool core. For logical inference, Z3 [9] SMT solver was chosen. Z3 is an open source SMT solver developed by Microsoft Research. It has an API for a vast set of programming languages. Nowadays, Z3 is considered as the most powerful tool for working with SMT predicates according to SMT-COMP results [23].

The evaluation was performed using different test programs. A total of 30 programs were used with a size of each program of 30-50 lines. Each program can contain or not contain defects. The same programs were analyzed with

Table I. ANALYSIS RESULTS

Analysis Tool	Precision	Soundness	Average Analysis Time
Prototype	95%	95%	53 sec
FlexeLint	60%	45%	4 sec
Aegis	68%	75%	5 sec

FlexeLint tool from Gimpel Software and with basic Aegis analyzer. The evaluation results are represented in the table I.

The given results show significant increase of analysis precision because of introducing dependency analysis rules. A drawback of the prototype is time consumption, which was from one second to 10 minutes for different programs.

Analysis of loops are the most time-consuming because a lot of predicates are created for each loop iteration. Several decisions exist which can increase performance. One of them is using of garbage collection. This method is assuming to remove all unnecessary predicates from a program state, so a predicate is removed if it is no longer in use. Another decision is simplification of a program state using predicate transformation rules. This approach allows to decrease number of external solver calls.

V. CONCLUSION

This article considers an approach for code analysis which is based on a combination of exact value analysis and predicate analysis together with logical inference tools. Predicate extraction rules are given for different program statements. Logical inference rules are build which provide pointer analysis and incorrect dereference detection. The given algorithms are implemented in the research prototype based on Aegis analyzer and Microsoft Z3 SMT-solver. A significant precision increase is shown comparing with the basic analyzer.

Future advancement directions are mostly related to methods for performance boosting, which can allow to use this approach for industrial program analysis.

REFERENCES

[1] "Coverity scan: 2012 open-source report," http://scan.coverity.com, 2012.

[2] V. Itsykson and M. Glukhikh, *Software engineering. Software quality assurance with code analysis methods (in Russian)*. SPb, Russian Federation: SPbSPU, 2011.

[3] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 672–681.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A fex billion lines of code later: Using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[5] P. Cousot, "Abstract interpretation," *ACM Computing Surveys*, vol. 28, no. 2, pp. 324–328, 1996.

[6] M. Glukhikh, V. Itsykson, and V. Tsesko, "Using dependencies to improve precision of code analysis," *Automatic Control and Computer Science*, vol. 46, no. 7, pp. 338–344, 2012.

[7] W. Rautenberg, *A Concise Introduction to Mathematical Logic*, New York, NY, USA, 2010.

[8] J. H. Gallier, *Logic for Computer Science. Foundation of Automatic Theorem Proving*, Philadelphia, PA, USA, 2003.

[9] "Z3 SMT solver," http://z3.codeplex.com/.

[10] P. Cousot, R. Cousot, and L. Mauborgne, "Theories, solvers and static analysis by abstract interpretation," *Journal of the ACM*, vol. 59, no. 6, 2012.

[11] "Static analysis tools for c code," http://spinroot.com/static, 2013.

[12] "Coverity save," http://www.coverity.com/products/coverity-save.html, 2013.

[13] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in system code," in *Symposium on Operating System Principles*. ACM, 2001, pp. 57–72.

[14] "The astree static analyzer," http://www.astree.ens.fr, 2013.

[15] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniax, and X. Rival, "Combination of abstractions in astree static analyzer," in *Proceedings of the 11th Annual Asian Computing Science Conference (ASIAN'06)*, Berlin, 2008, pp. 272–300.

[16] "The saturn program analysis system," http://saturn.stanford.edu, 2006.

[17] A. Aiken, S. Burgara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, "An overview of the saturn project," in *Workshop on Program Analysis for Software Tools and Engineering*. ACM, 2007, pp. 43–48.

[18] I. Dillig, T. Dillig, and A. Aiken, "Sound, complete and scalable path-sensitive analysis," *ACM SIGPLAN notices*, vol. 43, pp. 270–280, 2008.

[19] N. Nagappan and T. Ball., "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th international conference of Software Engineering*. ACM, 2005, pp. 580–586.

[20] W. Bush, J. Pincus, and D. Sielaff, "A static analyzer for finding dynamic programming errors," *Software: Practice and Experience*, vol. 30, pp. 775–802, 2000.

[21] "PC-lint and Flexelint static analysis tools," http://www.gimpel.com/html/products.htm, 2011.

[22] "Aegis static analysis framework," http://www.digiteklabs.ru/en/aegis/ platform, St. Petersburg, Russia.

[23] "Smt-comp 2012," http://smtcomp.sourceforge.net/2012/.