STATIC VALUE-RANGE ANALYSIS OVER C PROGRAMS

Daniela Ďuričeková
Master Degree Programme (3), FIT BUT
E-mail: xduric00@stud.fit.vutbr.cz

Supervised by: Tomáš Vojnar
E-mail: vojnar@fit.vutbr.cz

Abstract: In this paper, we propose a design of a value-range analyzer over C programs. First, we discuss a software vulnerability called buffer overflow and argue that value-range analysis may be used to prove the absence of buffer overflows. Then, an intuitive view of abstract interpretation, one of the approaches to value-range analysis, is given. Next, the Code Listener infrastructure, on which our analyzer is built, is described. After that, we discuss the design of the analyzer.

Keywords: value-range analysis, abstract interpretation, interprocedural analysis, Code Listener, buffer overflow, control-flow graph, call graph

1 INTRODUCTION

Arguably, one of the most well-known type of software vulnerabilities is a situation called *buffer overflow*. A buffer overflow occurs when data are written into a memory buffer that is not large enough to store these data. Buffer overflows may be exploited by a malicious person to gain control over the computer system. For example, in November 1988, an infamous Morris worm infected approximately 6000 network-connected hosts which represented 5–10% of the Internet at that time [2]. One of the primary replication mechanisms of the Morris worm was based on exploiting a buffer overflow in the `fingerd` daemon. However, in many cases, buffer overflow vulnerabilities do not need to be exploited by malicious persons to have disastrous consequences. Indeed, probably the best-known case of a buffer overflow is the Ariane 5 failure from 1996 where the catastrophe was caused by a program trying to store a 64-bit number into a 16-bit space [6].

Since both mentioned cases took place more than a decade ago, one might think that at present, buffer overflows are no longer an issue because programmers are well aware of them. However, the opposite is true. In the graph from Figure 1, the number of buffer-overflow-related errors in recent years is shown. Data for this graph are obtained from [1]. Of course, only reported errors are included. From this graph, it is obvious that the number of buffer overflow vulnerabilities is increasing.

Buffer overflows and other run-time errors, especially in critical software systems, may cause not only a loss of huge amounts of money but even worse—a loss of human lives. So, a need for a precise verification of the systems before their usage is consistently increasing. This need caused an emergence of formal verification methods. In summary, formal verification is the use of rigorous methods to ensure that a system conforms to some precisely expressed notion of functional correctness. There are several methods of formal verification and each of them is based on a different mathematical apparatus. Model checking, abstract interpretation, theorem proving and static analysis belong to the best known representatives of these methods [7].

To prove the absence of buffer overflows and other run-time errors, an analysis called *value-range analysis* in collaboration with data-flow analysis can be used. Both of them belong to static analysis techniques. Value-range analysis is based on arguing about the values that a variable may take on a given program point. For example, it may tell us that a variable `i` in the statement `a[i] = x;` can

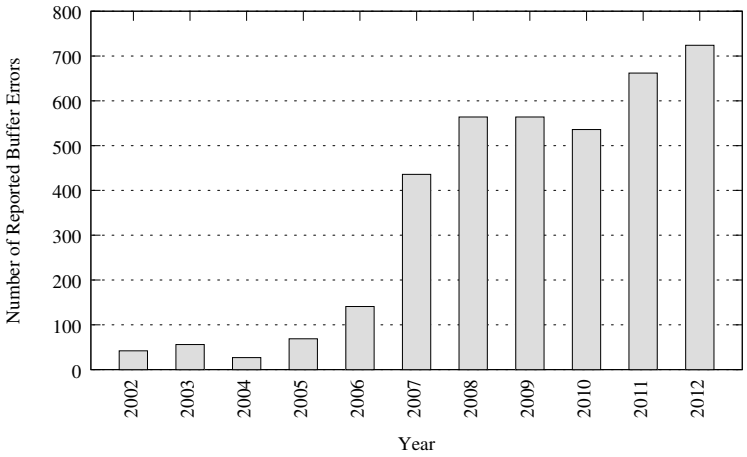

Figure 1: Number of reported buffer-overflow-related errors in recent years

be from the interval $[0, 10]$. This paper proposes the design of an interprocedural value-range analyzer for C programs.

2 DESIGN OF A VALUE-RANGE ANALYZER

There exist several approaches to value-range analysis. From all of them, abstract interpretation was chosen for our analyzer because probably the most successful value-range analyzer deployed in industry, namely the Value Analysis plug-in [3] from the Frama-C platform, is based on this approach. Informally, abstract interpretation is a static analysis technique that executes analyzed programs in an abstract way. During this execution, abstraction is used to preserve only important program properties and abstracts away all irrelevant details. This is done to speedup the execution and make it converge on infinite data domains, thus making the analysis computationally feasible.

Our analyzer is built on top of the *Code Listener infrastructure* (see [4]). Code Listener is a completely open-source infrastructure intended to simplify construction of tools for static analysis of C programs. Its long-term goal is wrapping the interfaces of existing code parsers and providing a unified, well-documented, object-oriented and easy-to-use Application Programming Interface (API) over these source code parsers. The aforementioned approach of using the existing code parsers to process source codes instead of leaving the job on static analyzers brings several advantages. Since code parsing is performed only once, we spare time as well as energy. Moreover, it is worth noting that every source code the parser is able to parse, the analyzer is able to use as its input. Therefore, static analyzers cannot fail due to problems with source code parsing. In addition, the Code Listener infrastructure provides a uniform interface for reporting errors. This means that errors are reported in the way programmers of static analyzers are accustomed from using the code parser, such as the GCC compiler (see [8]).

Our value-range analyzer receives program representation in the form of control-flow graphs (CFG) from Code Listener. A CFG (see [5]) is an oriented graph where nodes are basic blocks and edges follow the transfer of control. A basic block is a maximal sequence of consecutive statements. The first instruction in the basic block serves as the only entry point into that block. Similarly, the last instruction of the basic block is the only exit point from the block. Thus, there are no jumps into the middle of the block or from the middle of the block.

Our analyzer implements an interprocedural analysis based on [5]. Unlike intraprocedural analysis, interprocedural analysis is not restricted to a single function by ignoring function calls but it is typically performed across function boundaries. It means that the calling context of a function influences

data-flow information. For this reason, we need a representation of calling relationships between functions in the analyzed program named a call graph (CG). A CG (see [5]) is a directed graph where each node represents a function and edges represent calls between functions. This graph is also provided by Code Listener.

Intuitively, to compute the values variables may have in each block in the analyzed program, the analyzer has to use a suitable structure to represent these values. It may seem that it would be appropriate to use a set of values. However, this approach is very inefficient in terms of performance and fails for infinite domains. So, instead of a set of values, we use a union of intervals. We call this union a *range*. The analyzer is able to perform, for example, arithmetic, logical and comparison operations over ranges. Although that this approach brings a loss of accuracy, the gained results are an over-approximation of real results. Therefore, the value-range analysis is sound and efficient at the same time.

Our analyzer is implemented in the form of a GCC plug-in. So, it can be run by passing a special parameter to the GCC compiler, which specifies the used plug-in's name. After performing the analysis, for each function and each of its basic blocks, value ranges of each variable in the block are printed to the standard output.

3 CONCLUSION

In this paper, the design of a value-range analyzer for C programs was proposed. We believe that this analyzer will be useful for detecting buffer overflows and related errors in critical software systems.

ACKNOWLEDGEMENT

This work was partially supported by the BUT FIT grant FIT-S-12-1 and the research plan MSM 0021630528.

REFERENCES

[1] National vulnerability database. `http://nvd.nist.gov/home.cfm`. [cit. 2013-02-26].

[2] L. Boettger. The Morris Worm. `http://www.giac.org/paper/gsec/405/morris-worm-affected-computer-security-lessons-learned/100954`. [cit. 2013-02-26].

[3] P. Cuoq, V. Prevosto, and B. Yakobowski. Frama-C's value analysis plug-in. `http://frama-c.com/download/value-analysis-Oxygen-20120901.pdf`. [cit. 2013-01-21].

[4] K. Dudka, P. Peringer, and T. Vojnar. Code listener. `http://www.fit.vutbr.cz/research/groups/verifit/tools/code-listener/`. [cit. 2012-01-07].

[5] U. P. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.

[6] G. Lann. An Analysis of the Ariane 5 Flight 501 Failure—A System Engineering Perspective. `http://www.niwotridge.com/Resources/Ariane5Resources/78890339.pdf`. [cit. 2013-02-26].

[7] F. Nielson, H. R. Nielson, and Ch. Hankin. *Principles of Program Analysis*. Springer, 2005.

[8] GCC team. GCC, the Gnu Compiler Collection. `http://gcc.gnu.org/`. [cit. 2013-02-26].