Electronic Communications of the EASST
Volume 20 (2009)

Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2009)

Automatically Finding Bugs in Open Source Programs

Vladimir Nesov

10 pages

Guest Editors: Luis Barbosa, Antonio Cerone, Siraj Shaikh
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Automatically Finding Bugs in Open Source Programs

Vladimir Nesov

nesov@ispras.ru
Institute for System Programming of the Russian Academy of Sciences

Abstract: We consider properties desirable for static analysis tools targeted at finding bugs in the real open source code, and review tools based on various approaches to defect detection. A static analysis tool is described, that includes a framework for flow-sensitive interprocedural dataflow analysis and scales to analysis of large programs. The framework enables implementation of multiple checkers searching for specific bugs, such as null pointer dereference and buffer overflow, abstracting from the checkers details such as alias analysis.

Keywords: static analysis, security vulnerability, bugs

1 Introduction

Over the last decade, many tools for defect detection in source code of programs based on static analysis have been developed. Building on different analysis techniques, they are optimized for different usage scenarios, often inapplicable to the same task.

Static analysis tools may be used as a basis for program certification processes [BP08]. However, the nature of the open source development process makes it difficult to adopt many of the approaches employed by the existing defect detection tools. The need for the ease of adoption leads to a set of specific requirements for defect detection tools.

In this paper, we elaborate these requirements, and describe Svace, a static analysis tool that implements them. Svace is based on a novel scalable interprocedural program analysis algorithm, that allows to perform flow-sensitive dataflow analysis of 2,800,000 lines of code in 2.5 hours on one PC. The algorithm is implemented in a program analysis framework, that is used by a set of checkers, modules developed to search for specific kinds of defects in the source code of programs. The checkers use unsound analysis heuristics, and are developed using feedback from empirical evaluation of their performance on the analysis of a set of open source programs.

The rest of the paper is organized as follows. In Section 2, we develop a set of requirements for a static analysis tool allowing the ease of adoption in the open source context. In Section 3, we review different types of static analysis tools, and discuss areas of their applicability. In Section 4, we describe our static analysis tool. The paper is concluded in Section 5.

2 Requirements for practical open source defect detection tools

What makes a defect detection tool useful and easy to adopt for an open source project? There are several factors, but the central among them is the ability to *automatically* find new bugs.

Many open source projects have a large code base, which was not written with static analysis or testing in mind. Static analysis allows finding the violations of formal correctness rules without

setting up testing environment, on execution paths and data not covered by the tests. This allows using static analysis not only for finding obscure bugs that are hard to catch with testing, but also for finding bugs in poorly tested code.

Tools that check the source code of functions against their specifications are only as valuable as available specifications, and writing the specifications for the entire code base may be infeasible. The specifications written to catch one type of bugs using one tool will be unhelpful in catching other types of bugs using other tools. Defect detection tool should be able to search for bugs in the source code itself, without requiring additional information.

Results of static defect detection usually include false positives, issues that are reported, but are not really bugs. False positives are useless, and their prevalence in the report can significantly reduce the value of the static analysis tool [God05]. Limitations of analysis algorithms and lack of formal specifications of library or system functions lead to uncertainty, where reporting an issue risks creating a false positive, and skipping it risks missing a real bug. Static analysis tool should be able to distinguish issues that are likely to be real bugs.

Another reason for avoiding issues that are likely to be false positives is difficulty in manually assessing their correctness. Unlike a failed test, an issue reported by a static analysis tool cannot be directly examined with a debugger. The tool report a bug based on information gathered during analysis, and if that information is insufficient for the tool to be sure that the bug is real, the report may also be insufficient for programmer. Consider a warning about potential null pointer dereference, which doesn't explain how the dereferenced pointer can obtain a null value: maybe it is possible, and maybe not. When a potential bug depends on values passed interprocedurally across the whole program, the question may be hopeless.

Thus, the situations for which the static analysis tool is reasonably sure that they contain real bugs, are good for two reasons: they are likely to indicate real bugs, and the tool can explain its hypothesis in a report, so that it'll be possible for the user to check the correctness of the hypothesis. If the hypothesis is true, a bug is found, and if it's false, maybe there is a bug in that place for a different reason, but it falls outside the boundaries of issue reported by the tool.

To achieve precision and soundness, some static analysis methods require source code to satisfy certain restrictions. For example, complex data structures, C unions and recursion can be prohibited, and source code can be required to be presented in its entirety and use only known standard libraries. The focus on finding likely bugs and not on verification allows using unsound algorithms for analysis, which greatly simplifies the task of analyzing real programs without change.

Many bugs hiding in the code are interprocedural, and are caused by incorrect use of functions (including standard ones). Finding them requires interprocedural analysis. On the other hand, analyzing large programs requires analysis to be scalable. Summary-based analysis is a robust method for scalable interprocedural analysis. In summary-based analysis, each function of the program is assigned a summary, data structure of limited size that summarizes its behavior along dimensions important for analysis. Analysis of each function uses summaries of functions immediately called by it, and builds a summary for function itself.

Unsound summary-based analysis allows analyzing incomplete programs, which can, for example, use nonstandard libraries or system functions unknown to the tool. If a function uses inline assembler code that the tool can't analyze, it can locally exclude the analysis of the offending function.

Even if the analysis can proceed without specifications, having specifications for certain library or even program-specific functions can increase the quality of results. Some defects, such as format string vulnerability and SQL injection, are naturally formalized using taintedness property assigned to values, or using source-sink sequences. In either case, specific library functions will need to be given a specification, or be included in a property to be checked by the static analysis tool. In other cases, having a specification for a library function instead of incorrect default assumption may help to avoid false positives, or to not lose track of a bug. A certain non-library function in a large program can systematically confuse analysis algorithm, leading to big number of similar false positives. Thus, static analysis tool needs to provide a means of adding specifications where necessary, both for library and user functions, while trying to perform as good as possible without them.

3 Static defect detection tools

Static defect detection faces serious technical difficulties, and as a result, there are multiple specialized kinds of static analysis tools. These tools are targeted at their particular use cases, or limited by chosen technologies. Even though the properties listed in Section 2 seem to be desirable for most applications, many of the existing static analysis tools don't have them.

We will examine the existing tools, focusing primarily on tools for checking C programs, based on how they specify the situations in the source code that indicate the presence of bugs, and on the limitations imposed by algorithms for finding these situations in the source code.

Simple source code analysis tools, such as ITS4 [VBKM00], RATS and Flawfinder, are used to help with manual code audit. Such systems find certain template situations, such as potentially dangerous function calls, and list them exhaustively. Simplicity of analysis algorithms results in most of the reported issues not corresponding to real bugs.

Tools used to verify the absence of bugs of a certain type without requiring specifications usually impose restrictions on the source code of analyzed programs. Restrictions follow from inability of the existing sound analysis algorithms to work with arbitrary data structures (and, correspondingly, code constructions), and from the requirement to have correct, even if incomplete, information about library functions. One group of such tools in based on abstract interpretation techniques, and includes PolySpace (see comparison with other systems in [ZLL04]) and ASTRÉE [CCF+05]. These tools verify runtime safety and other properties of source code and were applied to check embedded software in aviation and device drivers. Another group of tools is based on counterexample guided abstraction refinement algorithm, which uses theorem proving to efficiently work with abstraction of state space. This group is represented by BLAST [HJMS02] and SLAM [BBC+06], applied to verification of runtime safety of device drivers. Limitations imposed by these tools make it difficult to apply them to regular open source programs, requiring either heavy revision of the source code, or isolation of parts of the code from incompatible features in preparation for checking.

Systems for user specification checking allow finding defects in more complex situations, and without restricting the source code, but require big number of manually written specifications to be effective. Cqual allows to add qualifiers to C types, and was used to find format string vulnerabilities [STFW01] and other bugs. LCLint [EGHT94], Splint [EL02] and CSSV [DRS03] use

user-written specifications to find buffer overflow and other defects. These tools may achieve verification of source code, proving the absence of certain kinds of bugs, if sufficient specification coverage is provided. However, as was mentioned earlier, providing the sufficient number of specifications may be infeasible for a large program, and specifications won't be useful for finding new kinds of bugs.

Automatic defect detection tools search for complex defects in source code of big programs, trying to satisfy the requirements listed in the introduction. One of the approaches to this problem is searching for a big number of specific templates of situations in source code, indicating possible bugs or violations of source code conventions. Such tools include FindBugs [HP04] and KLOCwork. The quality of analysis can be improved if a small number of user-written specifications is allowed. Splint [EL02] and ARCHER [XCE03] use unsound heuristics in a way that improves the quality of issues they report. MC [ECC01] (later developed into a commercial system Coverity) and Saturn [DDA07] look for local inconsistencies in the code, thus avoiding most of the false positives even given the imprecision in analysis. Commercial tools Coverity and CodeSonar find defects and security vulnerabilities using moderately general templates.

The diversity of approaches to static defect detection makes the comparison of static analysis tools very difficult. Tools belonging to different categories may be incompatible even where they emit warnings for the same bugs, due to different areas of applicability. There are usually only few tools falling in each category. The defect types detected by the different tools are often idiosyncratic, so that, for example, the kinds of situations in which the buffer overflow bug is detected by one tool are significantly different from those detected by the other tools. As a result, there are only few, mostly qualitative comparative studies, such as [ZLL04, EN08].

4 Our static analysis tool

We developed Svace, a static analysis tool for C programs (with limited support for C++) based on design choices presented in Section 2. Svace doesn't impose restrictions on the analyzed program, and collects its representation during execution of normal program build script, using a modified version of GCC compiler.

Svace consists of a framework that implements most of the work on program analysis, and a set of checkers that use that framework to implement simpler heuristic algorithms to search for specific defects.

The framework implements flow-sensitive interprocedural summary-based analysis. The summary-based analysis operates in a bottom-up fashion, as described in Section 2. Each function of the program is analyzed only once, using summaries of the functions called from it, and computing a summary of the function itself. The analysis doesn't use global information about the program, information is used interprocedurally only through application of the summaries of functions at their call sites. For a high-level review of various strategies for interprocedural analyses, see [CC02].

The analysis of each function proceeds on three levels. On the first level, Svace performs alias analysis [Hin01], associating abstract memory locations with the variables (names) of the program, including the variables accessed indirectly. The strategy for allocating abstract memory locations is similar to that used in [LH01]. Like in [LH01], the new identifiers are created lazily

where needed, but in Svace, the analysis in flow-sensitive, which requires some modifications (see [WL95] for a description of another related alias analysis algorithm). Since some of the memory locations are visible from multiple functions, the identifiers of memory locations are assigned locally, and translated during summary applications at call sites. Since each function is analyzed without information of its call sites, the alias information for the function parameters and global variables passed by the callers is unavailable. Following [LL03], we assume that the parameters of the functions are not aliased. This assumption is unsound, but empirically, it never introduced any problems for defect detection.

On a second level, Svace performs *value analysis*. The value analysis shares some characteristics and algorithms with def-use analysis or SSA analyses [CFR+91], but it has a different goal. Whereas SSA constructs new names, so that each name (variable) has only one definition, value analysis constructs identifiers for values that have only one definition. For example, if the same value is copied from one variable to another, it retains the same value identifier, and the same value identifier becomes shared by different abstract memory locations. Value identifier analysis allows to track equal values of variables along program execution paths.

Similarly to memory locations during alias analysis, value identifiers are assigned locally, and are translated between different namespaces during summary applications at call sites. This level of parameterization in summaries allows to approximate the effect of function calls at each call site in a context-sensitive manner [LH01], which improves the precision of analysis [Hin01]. At the same time, functions themselves are analyzed in a context-insensitive way, which guarantees scalability of the interprocedural analysis.

Function summaries parameterized by value identifiers allow, for example, to express and automatically extract a property of a function to return one if its arguments. At each call site, the return value will be assigned the same value identifier as that argument (value identifiers are local, different at each of the different callers). The alias analysis, running on the level below value identifiers, allows to do the same for dynamic data structures, where values are assigned through sequences of dereferences.

On a third level, the analysis framework allows to associate *attributes* with the value identifiers. The analysis framework implements propagation of attributes over control flow graph, within each function during intraprocedural analysis, and between functions using summaries. For each checker, we define necessary attributes and attribute propagation rules allowing to find situations we are looking for. These attributes track properties of values in the program, such as interval of possible integer values, possibility of being equal to null, or dependence on input from the network. This framework is analogous to the standard framework of dataflow analysis, but doesn't enforce the sound semantics on the attribute propagation procedures implemented by the checkers. Where possible, the checkers may implement sound semantics.

To summarize, the edges of the control flow graph of a function during its analysis, and the summaries of the function that were analyzed, are associated with three mappings: a mapping from names to abstract memory locations in the current namespace, a mapping from abstract memory locations to value identifiers, and a mapping from value identifiers to sets of attributes associated with them. Through the intermediary of value identifiers for pointers, and points-to attributes defined on them, these mappings also define a points-to graph.

To our knowledge, the use of flow-sensitive summary-based interprocedural dataflow (alias) analysis as a basis for an automatic defect detection tool (in the sense of Section 3) for C is novel,

```
1  int printf(const char *format, ...) {
2      char d1 = *format;
3      sf_use_format(format);
4  }
```

Figure 1: Example of library function specification

as is the use of summaries parameterized by value identifiers.

We search for specific bugs using pluggable checkers. Each checker recognizes a certain situation in the source code, which indicates the presence of a bug. Many checkers can detect the same type of bugs in significantly different situations. For example, buffer overflow bug is present in a statement that writes to a buffer by a constant index that is out of bounds, so checker for that situation can look at each statement and check if that's the case. At the same time, buffer overflow can result from using a return value from a library function as an index without checking if function returned a negative error code. Recognizing this situation requires knowledge about the function, and about the fact that value returned from that function is used as an index in buffer access. Thus, implementing different checkers requires gathering different information about the program, and may require additional library function specifications.

Another merit of searching for bugs using specific checkers is in documenting the results of analysis. Each checker has a description of situations that it tries to find, usual causes of false positives, ways of getting rid of false positives, and so on. Some checkers work on a given program well, and some don't.

Svace supports specifications of library and user functions in a form of stub implementations that use special functions to specify checker-specific attributes. For example, specification of standard function `printf` is implemented as presented in Figure 1. The dereference of argument `format` shows that argument is being dereferenced, which is used by checkers that search for null pointer dereference bugs. Special function `sf_use_format` works as a hook during analysis of specification, and allows format string vulnerability checker to set a `USE_FORMAT` flag on the formal argument `format`, a fact that is reflected in the summary of function `printf`, and then used by the same format string vulnerability checker when function `printf` is called in analyzed source code.

Let's consider, for example, a deference-of-null checker (only in intraprocedural case, to simplify the description). Dereference-of-null checker searches for situations in source code when dereferenced pointer can only have null value. Implementation of dereference-of-null essentially consists in constant propagation starting in conditional statements. A flag `IS_NULL` is propagated starting from the edge in the control flow graph where conditional resolves the value of pointer into null. If this flag meets a dereference operation (or a note in summary of called function stating that this value is dereferenced in it), warning is emitted. Our tool found dereference-of-null situation in 21 places in 14 open source programs we analyzed with this checker, and in 13 places there was a real bug. An example of this bug found in `zebra-0.94` is presented in Figure 2. Here, the last line can only be reached if structure pointer `rn` is equal to null, in which case it'll be dereferenced there.

```
1   if (rn)
2   {
3      route_unlock_node (rn);
4      zlog_info ("ifaddr_ipv4_add(): address %s is already added",
5         inet_ntoa (*ifaddr));
6      return;
7   }
8   rn->info = ifp;
```

Figure 2: Example of dereference-of-null bug

Table 1: Analysis results

Defect	Total	True
buffer overflow	23	8
C string error	13	10
null pointer dereference	119	59
TOCTTOU	32	30
chroot jail	9	5

Since the analysis is unsound, the development of checkers is guided empirically [HP04]. The checker development process starts from formulating the situations in source code that might indicate the presence of a certain kind of defect. The first implementation of the checker is done to find sufficiently many situations that potentially contain the bug, even if it results in a high false positive rate. Based on the test runs of the current implementation on a set of programs, the situation in source code sought by the checker is refined, to exclude as many of the false positives as possible, without excluding the real bugs. Where the incompleteness of library function specifications is found to confuse the checker, the library specification is expanded. These steps are repeated as necessary, to refine the checker. If the resulting checker becomes sufficiently accurate (in particular, on the analysis of the programs not tested during the refinement), it is included as a part of the tool, otherwise it retains the "experimental" status.

According to the classification given in Section 3, Svace is an automatic defect detection tool. The closest published tools are MC [ECC01] and Saturn [DDA07]. MC uses checkers, but its intraprocedural analysis is based on simpler attribute propagation techniques. Saturn performs summary-based analysis, but uses predicate abstractions and type inference, more theoretically loaded and heavy-weight techniques that make the analysis more computationally intensive and development of checkers more difficult, although the precision of such analysis is potentially greater.

We implemented several checkers for Svace, detecting bugs such as buffer overflow, null pointer dereference, format string vulnerability, double free, TOCTTOU. We tested the tool on a collection of 63 open source programs, total size of 2,800,000 lines of source code in C (including apache, putty, mysql, cvs, squid, ntp, openssh, lhttpd, bftpd, pound, ssmtp, troll-ftpd,

cfingerd, gzip, thttpd, telnet, vsftpd, qmail, tar, wu-ftpd, openftpd, sendmail, proftpd). The running time of analysis depends on the number of enabled checkers; with all checkers enabled, all 63 programs are analyzed in 2.5 hours, on one PC using one core of a Intel Pentium E2180, 2GHz processor.

Table 1 lists the results obtained on a subset of 33 of these programs of about 500,000 lines of source code, using a subset of checkers. The third column of Table 1 lists the number of true positives detected during analysis of these programs, determined by manual inspection of analysis results.

In 2006, via grant from US Department of Homeland Security, Coverity and Stanford used a restricted version of their static analysis tool to scan a significant number of open source projects, and have given access to the results to the developers of those projects. The number of defects per 1000 lines of code found by our tool (about 0.2 bugs/KLOC) is similar to that reported by Coverity in their open source static analysis project (0.1 to 0.5 bugs/KLOC, depending on analyzed program)[1]. KLOCwork is also known for publishing the bugs found by their tool in open source programs. Static analysis tools can be used to create certification processes, based on elimination of defects detected by those tools. For example, Coverity certifies a program as "Coverity clean", when its tool fails to find any defects in that program.

5 Conclusion

In this paper we considered the properties desirable for static analysis tools used to find bugs in real code. A tool should focus on situations in which it can accurately recognize bugs, trading soundness of analysis for relevance of results if necessary, it needs to allow specification of standard and user functions where possible without requiring it, it needs to avoid placing restrictions on admissible source code. Detecting more bugs requires interprocedural analysis, but the analysis needs to scale to large programs. Not all tools choose to follow these guidelines, as there are use cases requiring use of technologies incompatible with the requirements.

A static analysis tool was presented, that satisfies these requirements and performs defect search using a collection of checkers implemented in a framework for interprocedural attribute propagation, based on dataflow analysis.

Bibliography

[BBC+06] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. Mcgarvey, B. Ondrusek, S. K. Rajamani, A. Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.* 40(4):73–85, October 2006.

[BP08] P. Breuer, S. Pickin. Open Source Certification. In *2nd International Workshop on Foundations and Techniques for Open Source Software Certification*. Pp. 1–9. 2008.

[1] http://scan.coverity.com/rungAll.html – retrieved 24 Apr 2009

[CC02] P. Cousot, R. Cousot. Modular Static Program Analysis. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*. Pp. 159–178. Springer-Verlag, London, UK, 2002.

[CCF+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival. The ASTRÉE Analyzer. *Programming Languages and Systems* 3444/2005:21–30, 2005.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, K. F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13(4):451–490, October 1991.

[DDA07] I. Dillig, T. Dillig, A. Aiken. Static error detection using semantic inconsistency inference. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. Pp. 435–445. ACM, New York, NY, USA, 2007.

[DRS03] N. Dor, M. Rodeh, M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. *SIGPLAN Not.* 38(5):155–167, May 2003.

[ECC01] D. R. Engler, D. Y. Chen, A. Chou. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *Symposium on Operating Systems Principles*. Pp. 57–72. 2001.

[EGHT94] D. Evans, J. Guttag, J. Horning, Y. M. Tan. LCLint: a tool for using specifications to check code. *SIGSOFT Softw. Eng. Notes* 19(5):87–96, December 1994.

[EL02] D. Evans, D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software* 19(1):42–51, /2002.

[EN08] P. Emanuelsson, U. Nilsson. A Comparative Study of Industrial Static Analysis Tools (extended version). Technical report, Linköping University, January 2008.

[God05] P. Godefroid. The Soundness of Bugs is What Matters (position statement). In *BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools)*. 2005.

[Hin01] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. Pp. 54–61. ACM Press, New York, NY, USA, 2001.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Lazy abstraction. *SIGPLAN Not.* 37(1):58–70, January 2002.

[HP04] D. Hovemeyer, W. Pugh. Finding bugs is easy. *SIGPLAN Not.* 39(12):92–106, December 2004.

[LH01] D. Liang, M. J. Harrold. Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*. Pp. 279–298. Springer-Verlag, London, UK, 2001.

[LL03] B. V. Livshits, M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. *SIGSOFT Softw. Eng. Notes* 28(5):317–326, September 2003.

[STFW01] U. Shankar, K. Talwar, J. S. Foster, D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*. Pp. 16–16. USENIX Association, Berkeley, CA, USA, 2001.

[VBKM00] J. Viega, J. T. Bloch, Y. Kohno, G. Mcgraw. ITS4: a static vulnerability scanner for C and C++ code. In *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*. Pp. 257–267. 2000.

[WL95] R. P. Wilson, M. S. Lam. Efficient context-sensitive pointer analysis for C programs. *SIGPLAN Not.* 30(6):1–12, June 1995.

[XCE03] Y. Xie, A. Chou, D. R. Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC / SIGSOFT FSE*. Pp. 327–336. 2003.

[ZLL04] M. Zitser, R. Lippmann, T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*. Pp. 97–106. ACM, New York, NY, USA, 2004.