Static Testing: Static Analysis and Code Reviews

Important Tools for System Security

Jon Meyer & Ananth Jillepalli

July 31, 2017
Version 2.2

University of Idaho

CS 539: Applied Security Concepts

Summary

Modern software is complex and difficult, if not impossible, to exhaustively test. We do recognize that there are certain programing mistakes and constructs that contribute to many common types of bugs. Static Testing tools and techniques are a valuable part of identifying and eliminating these flaws, assisting in the development of secure and reliable software.

Contents

1 Objectives of this Tutorial >

1. Understanding Static Testing tools and techniques
   (a) Static Analysis Models and Tools
   (b) Code Walkthrough
   (c) Technical Review
   (d) Code Inspection
   (e) Using Code Review Checklists

1. Not coding is better than substandard coding. Because, one of the major causes behind software errors is poorly implemented code. The poor implementation of code is done because requirements are not carefully analyzed while implementing the code. No matter what which software development process is followed by a developer, there will always be bugs. Static Testing mitigates possibility of occurrence for most common of such issues [10].

2. Code Reviews are an efficient mechanism to root out some easily noticeable bugs from code through the use of different code review mechanisms. In this tutorial, we see a detailed explanation of what a "Code Walk-through" means and how process of code walk-through [26] occurs. There is also an explanation regarding relationship between the tendency of development groups and the benefits of this approach.

3. In this tutorial, an elaborate explanation of what a "Technical Review" means and what the process entails. Also included with the explanation is how a technical review [25] can be beneficial for inter-team review of code which makes the process of static testing between teams easier.

4. In addition, within the purview of this tutorial lies the explanation of what a "Code Inspection" [24] is and how the mechanism works out in the overall process of code review and static testing.

5. Last, but not least, the tutorial will also be taking an elaborate take at explaining what "Code Checklists" are, and how they can be used helpful in guiding any code review process, which is adopted by the team or assigned to be adapted by the team. Some standardized code checklists are mentioned and one challenge involves code review using a provided checklist [27].

2 Required Background <>

We assume that the reader of this tutorial has an extent of background knowledge in the following areas:

1. Working experience on usage of computers and software applications, software programming, and coding process.
2. Basic overall idea of programming languages and syntaxes.
3. Fundamentals of data structure programming modules like stacks, queue, linked list, etc.,
4. An overall idea on general issues like code efficiency, code optimization, etc.,

Due to restrictions on time and manpower resources, we are not able to make the ensuing tutorial to be completely self-contained from the perspective of a user. As such, the tutorial is best used when the user already has certain background skills and knowledge. The following are some areas where we expect the users of this tutorial to have some previous skills/knowledge:

1. Practical experience on using computers, installing and using common software applications. The tutorial does not explain anything regarding programming and coding process.
2. Similarly, the tutorial also does not explain how to declare functions/methods & classes and what syntax to be used for a specific programming language. The user of this tutorial is expected to have a general idea on such topics.
3. Fundamental knowledge of advanced programming concepts like data structures (particularly linked list, which is used in a challenge) is also preferable and would enrich the user's understanding of the tutorial.
4. Finally, a common knowledge on general programming issues like knowing what is code efficiency, code optimization, debugging of code, etc., would be very helpful for the user to gain the most out of this tutorial.

3 Hardware and Software Requirements <>

1. The tutorial contained within this document does not use any activity or challenge which involves a virtual machine or an operating system. All of the activities and challenges are static analysis of given code.

2. However, to use some of the static analysis tools which are mentioned in this tutorial, one needs a computer. In that case, a machine powerful enough to run a resource-heavy software (like MATLAB), without any major hindrances would be more than satisfactory.

4 Problem Statement: Development Mistakes <>

1. Human beings are not perfect.
2. Engineers, contrary to popular opinion, are human beings.
3. Ergo, engineers are not perfect (i.e., they make mistakes).
4. Attackers will try to exploit these mistakes, turning them into vulnerabilities.

1. Human beings are a species with inherently found imperfections, like any other biological species. These imperfections occur in a wide variety of instances, one of which could be imperfection in logical design or logical implementation. Such an implication gives cause to a scenario where in the aforementioned imperfect humans do something and that action becomes cause of a few shortcomings in the endeavor done by humans.
2. Unlike the popular opinion, engineers in general and software engineers in particular, are a part of human genus. Though there are attempts at automatizing the software coding process through artificial intelligence and adaptive coding, human engineers are here to stay for the coming future.
3. Since software engineers belong to human species as of now, these software engineers also have the same shortcomings as other humans. Therefore, software engineers have imperfections, which ultimately leads to prevalence of bugs (flaws) within the software, which is designed by the said software engineers. Code arising from improper code design can have bugs at any given time.
4. As long as there are bugs within the software, there would be malicious entities attempting to exploit the bugs and transform the exploited bugs into standardized vulnerabilities. Once standardized, these vulnerabilities contribute to majority of software exploitations worldwide.

(Theological arguments aside, so long as human beings remain imperfect, the systems we create will have flaws and people with malicious intent will search for them and attempt to exploit them.)

5 Proposed Solutions: Analysis and Review <>

Better development practices:

1. Careful requirements analysis
2. Sound design practices
3. Effective dynamic testing
4. Static analysis
5. Code reviews

1. One of the major causes of software errors is inadequate requirements determination [1]. Whether you're following a traditional waterfall-like process or a more "Agile" iterative process, at some point you need to determine what you're trying to build. If you don't understand what the final product should look like, you're unlikely to produce it.

2. Poor design is another major contributor to software errors [2]. Poor design leads to increased vectors of flaws/bugs in the system because a poor design will lead to even poorer implementation.

3. Effective dynamic testing is an important part of producing reliable software. Unfortunately, we cannot exhaustively test non-trivial programs [4]. Indeed, testing of non-trivial software can be reduced to the *halting problem* [15] for which, as we all know, there is no general mechanical solution [8].

4. Static Analysis is "a collection of algorithms and techniques used to analyze source code in order to automatically find bugs" [9]. Though these "algorithms and techniques" are often implemented in software tools, they can be executed manually.

5. Code reviews, or, more generally, code walk-throughs, reviews and inspections, are conducted by human beings with the intention of identifying bugs and error prone/unmaintainable code, including violations of best practices and coding standards. [10]

   Code Static Analysis and Code Reviews are often considered together under the heading "Static testing." Static testing refers to tools and techniques of validating software without executing it. [23] The remainder of this tutorial will concentrate on static testing.

6 Related News: Impact of Improper Coding <>

1. 11/12/2015 – Multiple buffer overflows in libpng [12].
2. 12/15/2015 – SQL injection vulnerability in Cacti 0.8.8f [13]
3. 2011 "Top 25 Most Dangerous Software Errors" [14]

1. Despite what we have seen previously in this class and elsewhere, well known software errors that should be avoided by all developers are still being encountered. Last November (2015) for instance, a buffer overflow vulnerability was found in libpng [12].
2. Only a month later an SQL injection vulnerability was found in host_new_graphs_save a part of Cacti 0.8.8.f (and, presumably, earlier versions) [13].
3. The fact that programmers keep making the same types of mistakes is demonstrated by the 2011 list of the top 25 "Common Weakness Enumeration" listing [14]. Buffer overflow and SQL injection vulnerabilities are numbers one and three on the previously mentioned list, and appeared on previous list versions, but we still encounter them.

7 Background: Static Testing <>

1. Static testing attempts to partially validate software without executing it. [23]
2. Typically divided into:
   (a) Static Analysis
   (b) Code reviews

Note: From my own testing experience, I can tell you that dynamic testing is expensive and impossible to do exhaustively. The fact that a tester or developer cannot exhaustively test a non-trivial program is well known by software testing and development community [8]

1. Static testing, via static analysis and code reviews, attempts to find common errors in design and implementation [23].
   (a) More on Static Analysis in slide 5.
   (b) More on Code Reviews in slide 9.

8 Background: Static Analysis <>

1. Static Analysis uses mechanical methods (i.e., software tools) to statically validate software [3].
2. Generally speaking, static analysis tools:
   (a) Operate without regard to what the code is intended to do [3].
   (b) Look for known poor programming practices [3].
   (c) Look for certain well understood programming errors [3].

1. Just as you can't prove software works through dynamic testing, static analysis/static testing in general, cannot completely validate non-trivial software [8] [15].
2. Typically, static analysis tools can help find potential problems in software without execution of the code. These problems are generally related to known poor, risky or sloppy programming practices [3]. Because executing the software is not required, static analysis can be conducted relatively early in the software development life cycle (SDLC), allowing errors to be caught and detected early.
3. With a generalization, all of the static analysis tools are usually capable of:
   (a) Carrying out static analysis without any concern or regard towards what actions, the code is intended to perform.
   (b) Inspecting the given code for commonly found poor programming practices.
   (c) Inspect the given code for previously documented programming errors.

Side Note: Static Application Security Testing (SAST) is a specialization of static analysis methodologies for security applications. To know more information, see [6].

9 Background: Static Analysis Models <>

1. Syntax and construct analysis [5].
2. Class structure and inheritance analysis [3].
3. State machine model analysis [3]
4. Control and data flow graph analysis [3]

1. Syntax and construct analysis is the most basic level of static analysis, which performs strict syntax checking (often much stricter than a compiler) along with a search for code idioms (common patterns), that are likely known sources of bugs [5].
2. Class structure and inheritance analysis is essentially an extension of syntax and construct analysis. Applied to object oriented languages, this looks at relationships between classes, looking for code idioms that are bug prone [3].
3. Ultimately, every program is a state machine. To the extent that code can be abstracted into a state machine and potentially be analyzed by static analysis tools to inspect for likely problems [7].
4. Using Control and data flow graph analysis, more advanced analysis tools can identify potential bugs, while similar techniques can be applied in code optimizations [20] [21].

Side Note: One challenging area for both static and dynamic testing is emergent behaviors. It is hard enough to test to make sure that code does what is intended. It is even more difficult to demonstrate that it does not do anything unintended. That is particularly true when you take two or more pieces of "working" functionality and integrate them into a single system. Often the "working" pieces of software interact in unexpected and undesirable ways. What would be helpful is, a way to demonstrate that code has the intended functionality but, *no other functionality.* For more information on an approach to doing just that see [22].

10 Background: Static Analysis Tools <>

1. Compilers / Interpreters / Assemblers [16]
2. Lint and variants (e.g., PC-LINT, Splint) [16]
3. Cppcheck [16]
4. Findbugs (for Java) [16]
5. Veracode [16]

1. Your compiler, interpreter or assembler is your second line of defense (you are the first) against errors in your code. They tend to have different configuration options available. For gnu compilers (e.g., gcc and g++), I recommend using: -Wall, which turns on most of the gnu warnings, and -Werror, which treats warning as errors. This forces programmers to address their compiler warnings before they can move on.
2. Lint and its variants apply more stringent C (and, sometimes, C++) syntax rules to identify poor and non-portable constructs that are likely to result in bugs. There are Lint like tools for other languages, but Lint's primary focus is pn the C family [17].
3. More extensive static analysis of C and C++ code can be conducted using Cppcheck, a freely available package under the GPL. Cppcheck includes checks for array bound overruns, unused functions in classes, uninitialized variables' memory/resource leaks and appropriate use of STL constructs [18].
4. FindBugs is a static analysis tool for Java. It is configurable, allowing users to write custom rules and detectors. Unlike tools like Lint and Cppcheck, rather than analyzing source code, FindBugs analyzes Java byte-code [19].
5. Veracode is a commercial analyzer intended to perform SAST. Veracode operates by analyzing application binaries for security issues without the need for source code availability [16].

11 Static Analysis Tools – Our Observations <>

1. There exist powerful tools for catching potential bugs and deviations from standard coding practices.
2. High rate of false positives.
3. Therefore, ignoring false positives leads to a maintenance nightmare.

These are my own observations from my experience with static analysis tools over the past thirty-five years:

1. Let's think about the simplest case. If your compiler or interpreter encountered a syntax error and, rather than producing an error or a warning it silently guessed what you meant and went on, it would be much harder to debug programs. Static analysis tools give you information about portions of your code that seem likely to have errors, allowing you to do something about it, without having to wait and find a bug in execution.
2. Analyzing code is hard. Even the best static tools can only point you to areas in your code that seem risky. Many of these findings will prove to be "false positives"; which means a code that's error free but fits some signature of a potential bug that particular tool uses. This is particularly true when you're writing new code that takes advantage of legacy code that may be "known" to work but that wasn't developed according modern standards.
3. There is a tendency for development groups to not want to spend time revising code to eliminate false positives. They either ignore them or turn them off in some way (different tools offer different mechanisms for this). Ignoring false positives can lead to either build output that is so flooded with them that a new warning af a real problem will be missed, or, if a warning is turned off (particularly if a warning is turned off globally), to the warning not appearing when it would be more meaningful.

12 Background: Code Reviews <>

1. Code reviews are a type of static testing that focuses on human review of code in an attempt to identify errors, particularly those that result from a misunderstanding of the design or from deviations from development standards. [23]
2. In order of increasing formality, reviews can be classified into three categories:
   (a) Code Walkthrough
   (b) Technical Review
   (c) Code Inspection

1. The type of static analysis which is carried out and led by humans is called as *Code Reviews*. Since code reviews are carried out by a group of humans, they are usually effective in weeding out design errors or poor development practices.
2. Depending upon several factors, code reviews are generally classified into three types. They are as follows:
   (a) **Code Walk-through**: More on Code Walk-through in slide 11.
   (b) **Technical Review**: More on Technical Reviews in slide 12.
   (c) **Code Inspection**: More on Code Inspection in slide 13.

13 Code Reviews – Common Features <>

1. Participants should be adequately prepared.
2. Ideally, preparation includes generating a list of findings up front. [24]
3. Review is led by a team member. [25]
4. A record is kept of issues found, which are tracked until closed. [26]

1. Review participants should be prepared. The degree of preparation varies with the formality of the review and the complexity of the code being reviewed. For a walk-through [26] of simple code, general familiarity with the task and the code base may be adequate. For a formal inspection of more complex code, significant study of the code and supporting document (e.g., requirements and design) may be needed.
2. Often reviewers will notice some of the same things. It can be helpful if a list of items can be circulated in advance so that a decision can be made about which can be put on the "findings" list without discussion and which need further discussion in the review meeting. [24]
3. Reviews are not free-forum meetings. Someone leads the process, though the leader varies with the type of review.
4. A list of "findings" or issues found is kept by a scribe, generally someone other than the leader of review team. This is to help assure items identified in the review are not forgotten. [25]

14 Background: Code Walkthrough < >

1. Developer led [26].
2. Peer reviewed [26].
3. Degree of formality varies [26].
4. Aim is as much team member education as defect hunting [26].

1. Code walk-through reviews are typically led by the person that wrote the code [26].
2. Members of the review team are typically from the developer's immediate work group [26].
3. The degree of formality can vary from "Hey, you guys got a couple of minutes to take a look at something for me?" to a scheduled meeting with assigned roles. In paired programming environments, the walk-through can even occur contemporaneously with writing the code [26].
4. Code walk-through reviews often serve the dual purposes of reviewing the code and familiarizing other members of the team with the code base [26].

15 Background: Technical Review < >

1. Led by a trained moderator [25].
2. Reviewed by peers and technical experts [25].
3. Well defined review process with varying degree of formality [25].
4. Broad range of purposes, including finding defects, evaluating alternative implementations and resolving technical issues [25].

1. Ideally, a technical review is led by a trained moderator who is not the code author, who is an experienced reviewer and developer who is very familiar with the review process, and knows how to keep it on track and make sure that important steps are not missed [25].
2. Technical reviews often include individuals from beyond the local team. For instance, the review team might be augmented by people with specific domain expertise or knowledge of regulatory concerns [25].
3. The degree of formality will vary, though you can generally expect technical review to be more formal and require more preparation than a code walk-through [25].
4. While walk-through reviews are often quite specifically about the code being reviewed, technical reviews more formally consider the broader system [25].

16 Background: Code Inspection < >

1. Led by a trained moderator [24].
2. Well defined formal review process with rules, checklists and entry / exit requirements [24].
3. Defined review team roles (e.g., Moderator, Recorder, Reader, Author) [24].
4. Formal follow-up process with time-lines [24].
5. Purpose to correct defects in the code and to find process improvements [24].

1. Code inspections are led by a trained moderator, who is an experienced reviewer and developer who is very familiar with the review process, and knows how to keep it on track and makes sure that important steps are not missed. This role is never taken up by code author [25].

2. Code inspection process is the most formal. There is generally a well defined formal process with procedures covering what needs to happen before, during, and after the inspection meeting(s). There are formal requirements for entry (e.g., supporting design documents reviewed approved) and exit (e.g., all review findings addressed and all unit tests passing with 100% code coverage) [25].

3. The major roles for code inspections should be clearly specified by the inspection process. For example, the moderator might be responsible for ensuring that all entry and exit requirements are met, for conducting the review meeting, for forwarding process improvement suggestions to the appropriate people and, ultimately, for approving the code [25].

4. Code-inspection processes typically include directions on how to follow-up on review findings and the expected time-lines for completion [25].

5. Code inspections are often part of a continuous improvement process. They look at ways in which the software development process can improve, as well as looking for defects in the specific code under review. For instance, if inspection teams are finding that similar errors are being made by different programmers, they might recommend changes in training or coding standards to help avoid those mistakes [25].

17 Code Review Checklists < >

1. Having a code review checklist can help guide your code review process.
2. There are many standardized code review checklists available on the web (e.g., [28], [29], [27], and many more).

1. A code review checklist is a good way to guide your code reviews, to help keep the review team focused and reduce the likelihood that your team will forget to look for common errors and pay the price of overlooking. If your team finds that certain types of errors are recurring in the team's code you should consider adding those errors to your review checklist.
2. Review checklists can vary from ones with a single page list to the ones with dozens of pages long lists (e.g., [28]). One should find a checklist that comes close to meeting the needs of one's team and use it as a base, revising the list as one works with it and come to a better understanding of what's best for one's team and its working environments.

18 Challenge: Technical Review < >

1. This challenge is intended to be a team effort, with ideally 3-4 people in the team*.
2. All members of the team, take five minutes to read through the National Weather Service checklist at [27]. (Also given in folder `Challenge`)
3. Using the check-list, work towards reviewing the code**. One person in the team, act as a moderator: to keep the process moving, and another person as a recorder/scribe. What defects does one see?

* But, the challenge can be taken up with one person acting as all the roles needed, though it would not give the same experience as corporate technical review process.

** The code file and the check-list are given in the folder `Challenge`. File with code is titled as `Code File` and checklist file is titled as `OHD-Checklist`.

19 Questions < >

1. In the `Code File` of the challenge, there was a potential for a major memory leak in the "UberNode". What was it?
2. What is the difference between Static Analysis and Static Testing?.
3. How do we know we can't write an algorithm to prove a non-trivial program is correct?

1. Since LinkedListNode does not have a virtual destructor, when an instance of UberNode is destroyed via a pointer to it's parent, UberNode's destructor will not be called. That means the destructors of UberNode's members (e.g., m_map) will not be called so they will not have a chance to free any memory they allocate.
2. Static Analysis is a subset of Static Testing. Static Testing involves using mostly mechanical methods to analyze software without executing it. Static Testing encompasses both Static Analysis and various forms of Code Review.
3. Trying to write an algorithm to prove that a non-trival program is correct is reducible to the "Halting Problem" witch has been proven (by Alan Turing) to be unsolvable (caveat, the proof was actually for a Turing Machine, which has infinite resources, not for a resource constrained physical computer) .

20 Conclusion < >

1. It is known that as long as human beings continue to write software, it is inevitable to avoid coming across bugs in our code, to some extent.
2. Many bugs follow recognizable patterns that can be detected by Static Testing, either by the mechanical methods of Static Analysis or by Code Review processes.
3. Proper use of Static Testing can help find and eliminate bugs that would be difficult and costly to remove using dynamic and automated mechanisms.

1. As derived in problem statement of this tutorial, human beings not being perfect creates a significant impact on design of a system and development of the needed software code. Therefore, as long as human beings continue to write software code, coming across bugs/flaws in software is inevitable to a certain degree.
2. That said, a wide variety of flaws/bugs occur because of relatively similar causes. These cause have reoccurring patterns, thus making them easily recognizable either by Static Analysis (manual and/or automated) or by Code Review mechanisms.
3. Well placed usage of static testing helps in making easier, the process of finding and eliminating commonly found flaws/bugs in a software code. The same goal can be achieved by dynamic and automated testing methods, but they are relatively costlier and difficult to do so, when compared with static testing.

21 Appendix: Solution and Change-log <

1. Solution to the challenge
2. Change-Log

1. **Solution to the challenge:** The solution to technical review checklist challenge is in the sub-folder named `Solution` in the folder `Challenge`.

   NOTE: However, it is to be noted that, due to the open-ended nature of technical reviews in general, the solution might not be reflective of what is absolutely right, but it acts a guide, using which, the user attains better understanding of good coding practices.

   NOTE 2: The given challenge deals with a very veteran and seasoned checklist. Thus, it brings out even the minute inconsistencies in the code. Subsequently, spending a lot of time on correcting the spacing, naming conventions of the code is a waste of time as there are tools to automate those kind of modifications.

2. **Change-Log:**

Static Testing: Static Analysis & Code Reviews Tutorial			
Ver.	**Date**	**Authors**	**Changes**
v1	Apr. 18th 2016	Jon Meyer	First draft of tutorial.
v2	Jun. 01st 2016	Ananth Jillepalli	Major content addition and remodeled the structure.
v2.1	Jun. 08th 2016	Ananth Jillepalli	Added appendix and other minor enhancements.
v 2.2	July 31st 2017	Ananth Jillepalli	Changed the licensing from CC BY-NC-ND 4.0 to CC BY-NC-SA 4.0

References

[1] Nelson, Mike et al. "Curing the Software Requirements And Cost Estimating Blues," *Project Management*. November-December 1999. http://www.dau.mil/pubscats/pubscats/pm/articles99/nelsonnd.pdf, downloaded 11 April 2016.

[2] Carlson, D. "Software Design Using C++ – Software Engineering," Saint Vincent College - Computing and Information Science Department, 31 January 2016. http://cis.stvincent.edu/html/tutorials/swd/softeng/softeng.html, downloaded 11 April 2016.

[3] Easterbrook, S. "Lecture 19: Static Analysis Tools," University of Toronto - Department of Computer Science, 2012. http://www.cs.toronto.edu/~sme/CSC302/notes/19-static-analysis.pdf, downloaded 17 April 2016.

[4] Kothari, J. *Topics in Software Testing: An Introduction.* https://www.cs.drexel.edu/~jhk39/teaching/cs576su07/L1.pdf, downloaded 11 April 2016.

[5] Hovemeyer, D. and Pugh, W *Finding Bugs is Easy.* https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0ahUKEwjRkMXV0J3MAhVGLmMKHQOJBgYQFggkMAE&url=https%3A%2F%2Fwiki.uta.edu%2Fdownload%2Fattachments%2F76972080%2Fruthwika.pptx%3Fversion%3D1%26modificationDate%3D1364344621000&usg=AFQjCNERx2ApnxCYcjuIGwoXtpokxSk6TQ&sig2=lo1-xLB2su3iZ7bq_klBYg, downloaded 17 April 2016.

[6] Brucker, A. and Sodan, U. *Deploying Static Application Security Testing on a Large Scale.* https://www.brucker.ch/bibliography/download/2014/brucker.ea-sast-expierences-2014.pdf, downloaded 17 April 2016.

[7] Saadatmand, M. and Cicchetti, A. "Mapping of State Machines to Code: Potentials and Challenges." *ICSEA 2014 : The Ninth International Conference on Software Engineering Advances.* http://www.es.mdh.se/pdf_publications/3668.pdf, downloaded 17 April 2016.

[8] *Halting problem.* https://en.wikipedia.org/wiki/Halting_problem, accessed online 11 April 2016.

[9] *Clang Static Analyzer.* http://clang-analyzer.llvm.org/, accessed online 17 April 2016.

[10] *Code Review.* http://www.tutorialspoint.com/software_testing_dictionary/code_review.htm, accessed online 17 April 2016.

[11] *Static Testing.* http://www.tutorialspoint.com/software_testing_dictionary/static_testing.htm, accessed online 17 April 2016.

[12] *CVE-2015-8126.* https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-8126, accessed online 17 April 2016.

[13] *CVE-2015-8377*. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-8377, accessed online 17 April 2016.

[14] *Common Weakness Enumeration*. http://cwe.mitre.org/top25/, accessed online 17 April 2016.

[15] *Static program analysis*. https://en.wikipedia.org/wiki/Static_program_analysis, accessed online 11 April 2016.

[16] *List of tools for static code analysis*. https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis, accessed online 17 April 2016.

[17] *Lint (software)*. https://en.wikipedia.org/wiki/Lint_(software), accessed online 17 April 2016.

[18] *Cppcheck*. https://en.wikipedia.org/wiki/Cppcheck, accessed online 17 April 2016.

[19] *FindBugs*. https://en.wikipedia.org/wiki/FindBugs, accessed online 17 April 2016.

[20] *Control flow graph*. https://en.wikipedia.org/wiki/Control_flow_graph, accessed online 17 April 2016.

[21] *Data-flow analysis*. https://en.wikipedia.org/wiki/Data-flow_analysis, accessed online 17 April 2016.

[22] De Leon, D.; Alves-Foss, J. and Oman, P., "Implementation-Oriented Secure Architectures,". textit2007. 40th Annual Hawaii International Conference on System Sciences, Waikoloa, HI, 2007, pp. 278a-278a. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4076945&isnumber=4076362, downloaded 17 April 2016.

[23] *Static Testing*. http://www.tutorialspoint.com/software_testing_dictionary/static_testing.htm, accessed online 17 April 2016.

[24] *Code Inspection*. http://www.tutorialspoint.com/software_testing_dictionary/code_inspection.htm, accessed online 17 April 2016.

[25] *Code Review*. http://www.tutorialspoint.com/software_testing_dictionary/code_review.htm, accessed online 17 April 2016.

[26] *Code Walkthrough*. http://www.tutorialspoint.com/software_testing_dictionary/code_walkthrough.htm, accessed online 17 April 2016.

[27] "C++ Coding Standards and Guidelines Peer Review Checklist," 16 November 2006. http://www.nws.noaa.gov/oh/hrl/developers_docs/OHD_C++_Programming_Peer_Review_Checklist.doc, downloaded 17 April 2016.

[28] "An Abbreviated C++ Code Inspection Checklist," 27 October 1992. `http://www.literateprogramming.com/Baldwin-inspect.pdf`, downloaded 17 April 2016.

[29] *Code Inspection Rules.* `http://www.oualline.com/talks/ins/inspection/c_check.html`, accessed online 17 April 2016.