QUALITY CODE ANALYSIS IN STUDENTS' PROJECTS

Daniela GOTSEVA, Ognian NAKOV, Luka BEKIAROV
Technical University of Sofia

Abstract. In this study we focused on three ASA tools for C/C++ programs. After short ASA tools description, we examine the faults identified by ASA tools, manual inspections and system failure testing. Additionally we categorize raw output from ASA tools that help us to make conclusions about the efficiency of static analysis for software fault detection in students' projects. We analyze 500 student projects during the last 12 years.

Key words: static analysis, bug reviews, C/C++ and Java programming

1. Introduction

One of the possible fault-detection techniques is static analysis. This analysis concerned evaluating a system and its components based on a code, forms, data structures, documentation without program execution. Inspections are an example of static analysis that relies on code rewires. The other possibility is using of automated tools for that purpose. These tools help us to reduce code errors such as runtime exception, redundant code, inappropriate use of variables, division by zero and potential memory leaks. We defined the use of *automated static analysis* (ASA) tools and *Inspections* that mean manual code review. ASA may help software engineers to fix faults in software test process. In this paper we report the result of using static analysis procedures as a fault detection technique in students' projects.

The study was a research that analyzed 500 students' projects in Technical University of Sofia, Computer System Department. Since 1996 we collect, inspect and analyze by ASA tools over 9 million lines of code (LOC). In our research we examine software projects written in C/C++ that underwent various combinations of inspection and ASA. We used Goal-Question-Metric (GQM) to motivate and focus our data collection and analysis.

2. Background

In this section we provide an overview of ASA tools and classes of faults and failures that are most often detected by ASA and by inspectors.

2.1. Automated Static Analysis Tools

ASA can be used as an added fault-detection filter in software development process. ASA tools automate the identification of certain types of anomalies, as discussed above, by scanning and parsing source text of a program to look for a fixed set of patterns in the code. ASA utilizes control flow and data flow analysis, interface and information flow analysis of the source code. There are some errors that are never detected by ASA tools [5, 6]. Additionally every ASA tool generates different, sometimes no overlapping, errors [7].

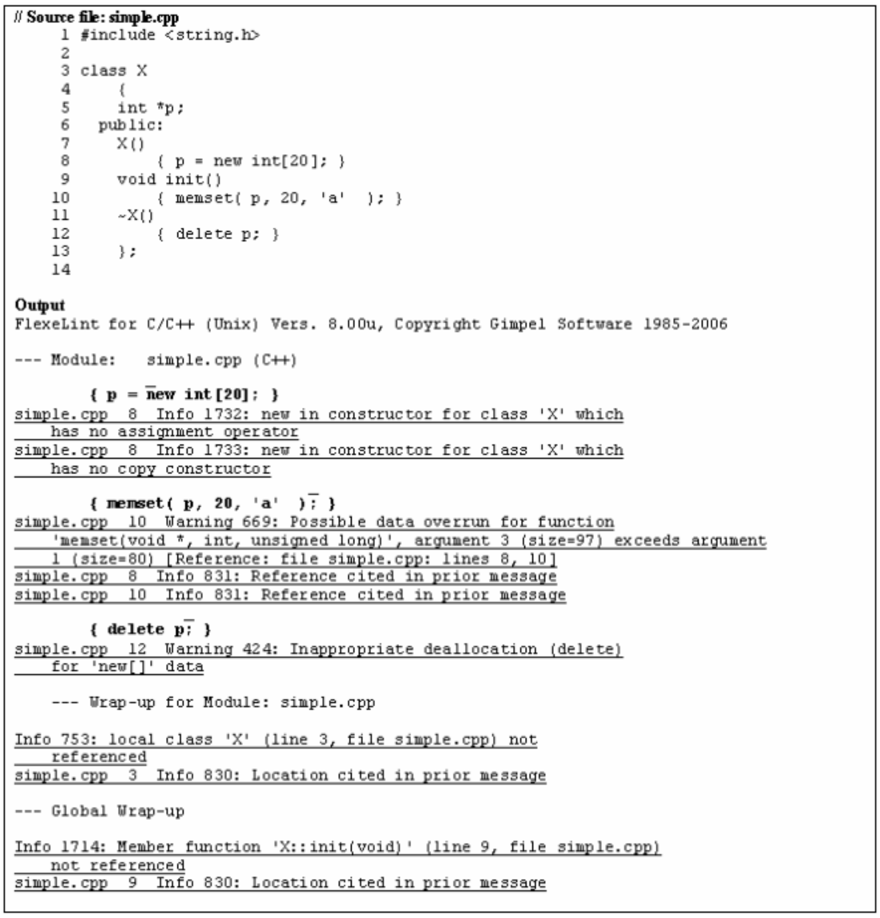
The important benefit of ASA is that they do not require code execution for bug tracking. In this case ASA is opposite to the language compilers. C language doesn't have strong type checking and the compiler can omit some errors. They can be trapped by ASA tools.

There are range of ASA tools and services deployed for C/C++ programs. One of these products is FlexeLint [1]. It will check C/C++ source code and find bugs, glitches, inconsistencies, non-portable constructs, redundant code, and much more. It looks across multiple modules, and so, enjoys a perspective your compiler does not have. FlexeLint is a Unix-based tool and there is also Windows-based version: PC-Lint. The result of the tool working is demonstrated on figure 1.

The other good ASA tool is Reasoning [2]. Its services boost the productivity of development teams by uncovering security vulnerabilities and reliability defects before they become costly problems. This tool finds defects in C/C++ applications. Reasoning's Discovery Mapping Analytics Service (DMA) is an analysis of users' source code using static analysis techniques and Reasoning's expertise in identifying Implementation Defects. The tool process users' code through various static analysis engines and analyze the results to benchmark the quality level of each component of the application.

The followed metrics are embedded into Reasoning:

- Metrics for Prioritization – Using Reasoning's Discovery Mapping Analytics service, the



// Source file: simple.cpp

```
     1 #include <string.h>
     2
     3 class X
     4     {
     5     int *p;
     6   public:
     7     X()
     8         { p = new int[20]; }
     9     void init()
    10         { memset( p, 20, 'a'  ); }
    11     ~X()
    12         { delete p; }
    13     };
    14
```

Output

```
FlexeLint for C/C++ (Unix) Vers. 8.00u, Copyright Gimpel Software 1985-2006

--- Module:   simple.cpp (C++)

        { p = new int[20]; }
simple.cpp  8  Info 1732: new in constructor for class 'X' which
    has no assignment operator
simple.cpp  8  Info 1733: new in constructor for class 'X' which
    has no copy constructor

        { memset( p, 20, 'a'  ); }
simple.cpp  10  Warning 669: Possible data overrun for function
    'memset(void *, int, unsigned long)', argument 3 (size=97) exceeds argument
    1 (size=80) [Reference: file simple.cpp: lines 8, 10]
simple.cpp  8  Info 831: Reference cited in prior message
simple.cpp  10  Info 831: Reference cited in prior message

        { delete p; }
simple.cpp  12  Warning 424: Inappropriate deallocation (delete)
    for 'new[]' data

    --- Wrap-up for Module: simple.cpp

Info 753: local class 'X' (line 3, file simple.cpp) not
    referenced
simple.cpp  3  Info 830: Location cited in prior message

--- Global Wrap-up

Info 1714: Member function 'X::init(void)' (line 9, file simple.cpp)
    not referenced
simple.cpp  9  Info 830: Location cited in prior message
```

Figure 1. Simple C++ example and FlexeLint output

engineering managers will have the information to recognize and prioritize the most problematic application modules and direct engineering resources for improved effectiveness and more predictable results.

- Metrics for Risk Management – Reasoning's Discovery Mapping Analytics service maps implementation errors that cause system crashes or security vulnerabilities. With this mapping, the user can significantly enhance risk assessment and management capability, with improved prioritization, calibration and predictability. Improvements in process for discovering defects will improve the ability to manage the risk of killer defects getting to customers.
- Metrics for Code Integrity Benchmarking – Reasoning's Discovery Mapping Analytics service is a unique addition to your management dashboard, providing quality benchmarks. Reasoning's DMA service provides an unbiased, third-party assessment of code reliability and vulnerability. The DMA results show how users' code characteristics compare to those of some of the world's largest development organizations. Once measured, the user can compare: results of a quality initiative over the lifecycle steps of an application, use of different methodologies, team structures, training protocols, etc.

Reasoning DMA service provides metrics, giving the dashboard measurements that let manage quality initiatives effectively. Some of the possible error tracking with Reasoning are: NULL pointer assignment, out of array access bounds, memory leaks, bad deallocation and uninitialized variables.

The last tool present in our work is Klocwork K7 [3]. Its static analysis (see figure 2) is the most mature, scalable solution on the market, enabling accurate and efficient defect detection early in user process. K7 can be deployed both at the developer desktop and at a full system build time, ensuring full coverage of users code problems and enabling integration static analysis best practices in a way that best suits your development processes.

Klocwork K7 has two main functions:

➢ **Operational Defect Detection and Removal**

K7 provides user's development team with a multi-dimensional analysis of fundamental defect categories. It is designed for ease-of-use build-over-build with features such as a build and report

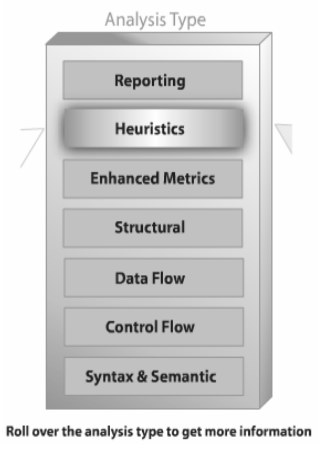


Figure 2. Klocwork static analysis technology

management GUI, industry leading message filtering, flexible configuration, and the powerful learning and tuning knowledge base. To do this, K7 analyzes C, C++, and Java source code and provides a summary of code problems, including defects, security vulnerabilities, architectural foundation issues, and metrics. It highlights the critical problems and separates them from the non-critical problems (see figure 3), allowing the team to assess the reported issues and decide whether to fix them or filter them. See table 1 for list of defect categories.

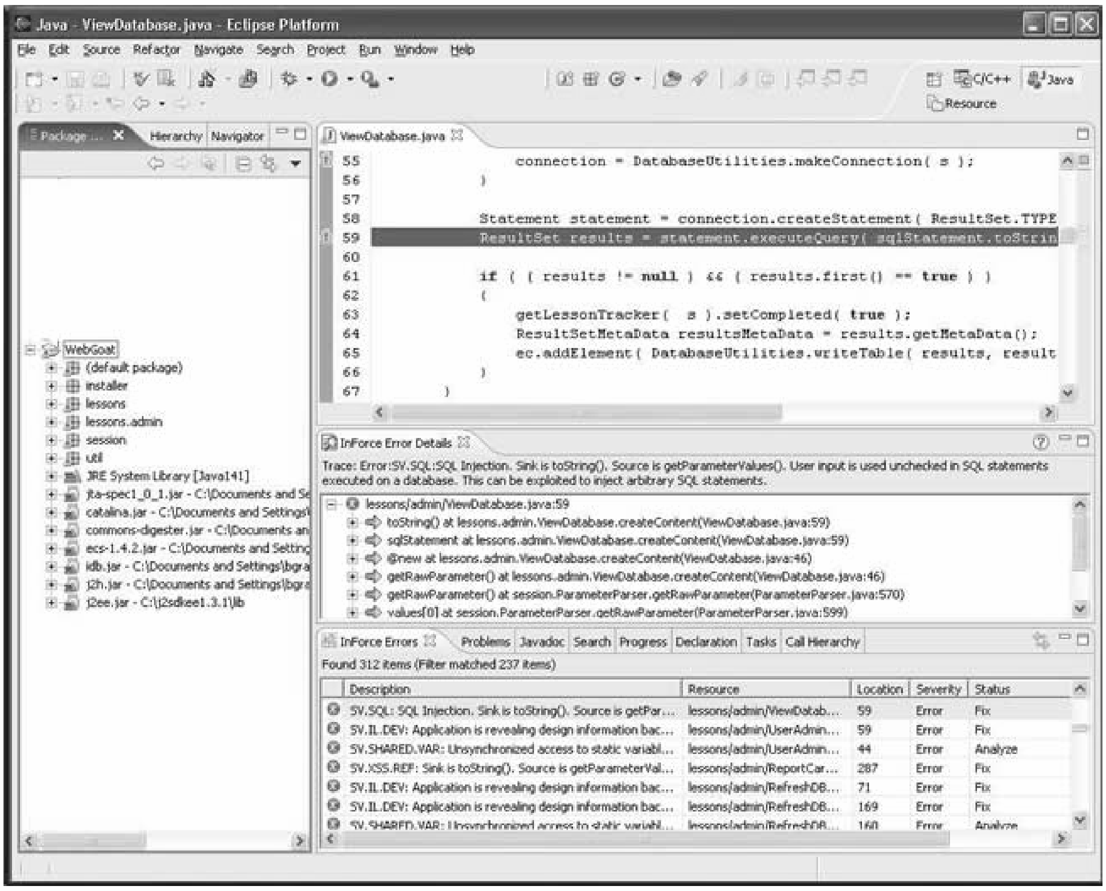
➢ Defect Detection and Prevention

The challenge for organizations that build or maintain large applications is to identify defects as early as possible in the development cycle (see figure 4). A defect that is identified and corrected while the developer is writing the code has a cost that is orders of magnitude lower than correcting the same defect in any of your existing test phases or once the product is released.

For years, the benefits of finding software

Table 1. Defect Categories List for C/C++ and Java languages

Sample C/C++ Defect Categories	Sample Java Defect Categories
Null Pointer Dereference	Efficiency Errors (e.g. Empty finalize method)
Memory Management Problems (e.g. Memory Leak)	Maintainability (e.g. Empty catch clauses)
Array Bounds Violations	Reliability (e.g. Resource Leaks)
Use of Uninitialized Data	
Coding Style Issues (e.g. Assignment in Condition)	



```
WebGoat
    (default package)
    installer
    lessons
    lessons.admin
    session
    util
    JRE System Library [Java141]
    jta-spec1_0_1.jar - C:\Documents and Se
    catalina.jar - C:\Documents and Settings\
    commons-digester.jar - C:\Documents an
    ecs-1.4.2.jar - C:\Documents and Setting
    idb.jar - C:\Documents and Settings\bgra
    j2h.jar - C:\Documents and Settings\bgra
    j2ee.jar - C:\j2sdkee1.3.1\lib
```

```
                connection = DatabaseUtilities.makeConnection( s );
            }

            Statement statement = connection.createStatement( ResultSet.TYPE
            ResultSet results = statement.executeQuery( sqlStatement.toStrin

            if ( ( results != null ) && ( results.first() == true ) )
            {
                getLessonTracker( s ).setCompleted( true );
                ResultSetMetaData resultsMetaData = results.getMetaData();
                ec.addElement( DatabaseUtilities.writeTable( results, result
            }
        }
```

```
Trace: Error:SV.SQL:SQL Injection. Sink is toString(). Source is getParameterValues(). User input is used unchecked in SQL statements
executed on a database. This can be exploited to inject arbitrary SQL statements.

lessons/admin/ViewDatabase.java:59
    toString() at lessons.admin.ViewDatabase.createContent(ViewDatabase.java:59)
    sqlStatement at lessons.admin.ViewDatabase.createContent(ViewDatabase.java:59)
    @new at lessons.admin.ViewDatabase.createContent(ViewDatabase.java:46)
    getRawParameter() at lessons.admin.ViewDatabase.createContent(ViewDatabase.java:46)
    getRawParameter() at session.ParameterParser.getRawParameter(ParameterParser.java:570)
    values[0] at session.ParameterParser.getRawParameter(ParameterParser.java:599)
```

Found 312 items (Filter matched 237 items)

Description	Resource	Location	Severity	Status
SV.SQL: SQL Injection. Sink is toString(). Source is getPar…	lessons/admin/ViewDatab…	59	Error	Fix
SV.IL.DEV: Application is revealing design information bac…	lessons/admin/UserAdmin…	59	Error	Fix
SV.SHARED.VAR: Unsynchronized access to static variabl…	lessons/admin/UserAdmin…	44	Error	Analyze
SV.XSS.REF: Sink is toString(). Source is getParameterVal…	lessons/admin/ReportCar…	287	Error	Fix
SV.IL.DEV: Application is revealing design information bac…	lessons/admin/RefreshDB…	71	Error	Fix
SV.IL.DEV: Application is revealing design information bac…	lessons/admin/RefreshDB…	169	Error	Fix
SV.SHARED.VAR: Unsynchronized access to static variabl…	lessons/admin/RefreshDB…	160	Error	Analyze

Figure 3. Integrating K7 into Eclipse IDE

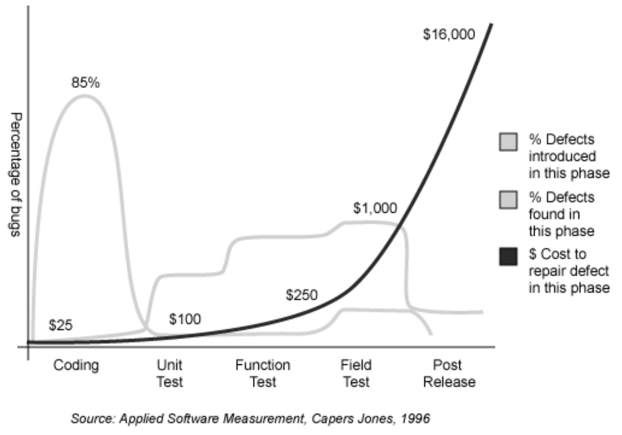


Figure 4. Area of bug tracking

defects before QA have been well documented and understood. Unfortunately, the only mechanisms available to do this efficiently have been either manual code reviews and/or lint-type static code analyzers. Both these approaches do have their benefits but suffer from various inefficiencies, least of which is an ability to establish good coverage of a large code base.

Now, with the advancement of static code analysis technology, potentially catastrophic defects can be found automatically and accurately. These problems will be identified early in the development process since static analysis does not require running code – it can operate even before user compile and integrate software. The removal of these problems improves the quality of software, while enhancing developer productivity.

In this study we used Basil taxonomies as appropriate for students' project assessment.

3. Case Study Details

In this section we explain how to collect the students work and some limitation to our research.

3.1. Data Collection

We collected and analyzed faults into 500 students' projects during last twelve years. Data analysis consists of faults for above nine million LOC written in C/C++, developed by bachelor and master of computer science students. In our analysis we discussed three cases: projects with inspections, projects with ASA tools testing at first step and inspections at second, and projects with inspections first and ASA tool testing after that. For our research purpose we used different ASA tools, as shown in table 3. In this study the number of errors found by FlexeLint is two times than errors found by Klocwork K7, and four times than errors found by Reasoning. Therefore we based our

Table 3. Data Analysis

Project Version	ASA	Inspections
SP1.0	Not performed	Yes
SP1.1	FlexeLint, Reasoning	Yes
SP1.2	FlexeLint, Klocwork	Yes

analysis on FlexeLint results.

We denoted students' projects as SP and defined three product versions: SP1.0 that have only handle inspections, SP1.1 and SP1.2 that are different in ASA tools but have ASA tool testing and inspections too.

3.2. Limitations

There are some limitations in our research. We used ODC taxonomies to classify errors. Next classification is done without information about severity or impact of potential failure. Additionally our study focused only three ASA tools and isn't representative of all ASA tools. Finally, the result isn't representative for programmers fault and it may be different for projects, created in other languages than C/C++.

4. Results

In this section we provided the achieve results. We divided them into five categories, given in sections 4.1 - 4.5. The basic goal is to determine whether ASA tools can help to students to improve their programming techniques and to see what kind of errors are most frequently occurred in students' projects. Each section started with a base question and short explanation of a used metrics to answer to the question in it. Using of GQM in each section help us to collect and analyze data. All of data analysis, implication of it are posted and discussed.

4.1. Student's Project Quality

The question that is important in this section is: "*Will a student's project be of higher quality if ASA tool is using in development process?*"

To answer to this question we used:

- quantity of defects found by system testing;
- quantity of defects found by our testing.

Divided by churned thousand lines of code (KLOC). The results were shown in table 4. Project quality comparison based on a number of total failures per churned KLOC (KLOCC). In the table we used SP1.0 as a baseline project for comparison. We normalized the failures per KLOCC metrics relatively to the SP1.0 projects. This gives us relative quality of SP projects.

There is a wide variance in the relative quality

Table 4. Relative SP Projects Quality

Project	Relative Quality (failures/ KLOC$_C$)	Process step 1	Process step 2
SP1.0	1.0	Inspections	
SP1.1	1.32	ASA	Inspections
SP1.2	0.41	Inspections	ASA

of the projects. As a result, ***our analysis didn't provide conclusive results about whether ASA tools will help to increase the SP projects quality***.

4.2. Fault Detection Yield

The question that we asked in this section is: "*How effective is ASA at detecting faults compared with inspections and testing?*"

To answer to this question we used:

- quantity of ASA faults;
- quantity of inspection faults;
- quantity of test failures.

Fault detection yield (FDY) refers to the percentage of defects, present in the code at the time of fault detecting practice [10]. FDY can't be precisely computed until the project is used extensively by the users. This measure decreased as more bugs are found. Additionally we calculated defect removal efficiency (DRE) [11] as a measure of how well bugs are removed. Software defect removal efficiency is percentage of total bugs eliminated in the code. High level of defect removal efficiency is corresponding to high level of user satisfaction.

For SP1.2 ASA performed during test, for SP1.1 ASA performed prior to inspection. No ASA test is done for SP1.0. The results are shown in table 5. For project SP1.2 the faults detections yield of test is relatively low, because ASA was performed during test. However DRE for project SP1.2 is 99.5%, which is approximately the same as the other projects. Research indicates that the user can receive high quality project, if the result is greater than 95% [11, 12]. The value of DRE is higher than industrial benchmark and this fact indicated high quality of software project.

These results indicate that defect removal yield of ASA isn't significantly different from that of inspections. The defect removal yield of execution-based testing is two times higher than that of ASA and therefore may be more effective at finding the defects.

4.3. Classes of Faults and Failures

The main questions discussed in this section are: "*What classes of faults and failures are most often detected by ASA, by inspection and by system testing? What classes of defects are escaped to the customers?*"

To answer to them we used:

- quantity of ASA faults by ODC type;
- quantity of inspection faults by ODC type.

We counted faults according to ODC type classification. Here we will present the results from each metrics.

4.3.1. ASA Faults

Each fault was documented with a problem explanation and detailed information such as: description, location, precondition, impact, severity, suggestion and code fragment. Then every fault was manually classified according to ODC types. Finally faults were counted and percentages are calculated. A summary of the results are shown in table 6. Only FlexeLint is included in comparison.

Table 6. ASA faults classification according to ODC taxonomies

ODC taxonomy	SP1.1 (%)	SP1.2 (%)
Assignment		
■ All tools	80	80
■ FlexeLint including	61	78
Checking		
■ All tools	20	20
■ FlexeLint including	50	25
Other ODC taxonomies	0	0

The result shown in table 6 indicated that ASA tools are effective for identifying two ODC types: Assignment and Checking. Checking defects are happen in low level design or coding phases and Assignments were occurred only in coding phase. These problems are due to logical than static analysis.

4.3.2. Inspection Faults

All inspection faults are documented in text file. Every inspection file was manually created and classified according ODC types. The result of this classification is shown in table 7. Note that in handle inspection some additional properties are documented: readability of code, maintainability, naming convention, coding standards and

Table 5. Defect Removal Yield

Project	Phase ASA	ASA (%)	Inspections (%)	Test (%)	DRE (%)
SP1.0	Not performed	Not preformed	42.31	96.73	98.10
SP1.1	Prior to inspections	35.00	20.48	98.18	99.05
SP1.2	During test	37.50	33.21	62.57	99.50

Table 7. Inspection faults classification

Defect Type	SP1.0 (%) No ASA	SP1.1 (%) After ASA	SP1.2 (%) Prior to ASA
Algorithm	30.40	38.12	38.27
Documentation	29.01	35.13	25.12
Checking	26.62	17.85	17.98
Assignment	6.33	5.02	8.26
Function	1.26	1.74	1.24
Interface	2.21	1.02	0
Build/Package/Merge	4.17	1.12	9.13

programming style. These comments are approximately 25% of statements in inspection records. They are not included in ODC taxonomies.

The results show that inspection identifies Algorithm, Documentation and Checking faults. Approximately 85% of all faults belong to these three categories and the distribution is constant regardless of whether or not ASA tests are performed.

4.4. Programmer Errors

The questions discussed in this section are: "*What kind of programmer errors is most frequently identified by ASA? How often does ASA find these errors?*"

To answer to them we used:

- quantity of ASA faults by defect type

To avoid differences in defect types among different tools, only one ASA is used. We choose FlexeLint, because it identified most defect types from examined ASA tools. Then we merged the same and very similar static analysis fault to performed result aggregation. The result is shown in table 8. All data are ranked with most frequently faults at the top of the list. FlexeLint can detect

Table 8. Detailed classification of static analysis faults ordered by Total occurred time

Fault Description	Critical (%)	Major (%)	Minor (%)	Total (%)	ODC classification
Possible use of NULL pointer	11.91	14.73	19.28	45.92	Assignment
Possible access Out-of-Bounds	0.49	3.46	6.18	10.13	Checking
Pointer has not been freed or returned	1.04	6.87	0.20	8.11	Assignment
Memory leak	2.92	3.76	0.79	7.46	Assignment
Variable not initialized before using	0.30	0.89	4.45	5.64	Assignment
Inappropriate deallocation	0.74	1.88	0.79	3.41	Assignment
Suspicious use of	0.10	0.35	2.03	2.47	Checking
Data overrun	0.05	0.15	1.93	2.13	Checking
Type mismatch with switch expression	0.10	1.93	0.15	2.18	Checking
Control flows into case/default	0.05	0.69	1.68	2.42	Checking
Possible passing a null pointer to function	0.35	0	1.04	1.38	Checking
Ignore return value of function	0.10	0.84	0.40	1.33	Assignment
Passing NULL pointer to function	1.09	0	0	1.09	Checking
Unusual use of a Boolean	0	0.54	0.54	1.09	Checking
Pointer member neither freed nor zero'ed by destructor	0	0.94	0	0.94	Assignment
Loop not entered	0	0.20	0.59	0.79	Checking
Unreachable code	0	0.30	0.49	0.79	Checking
Boolean argument to relational	0	0.30	0.05	0.35	Checking
Unparenthesized parameter	0	0	0.35	0.35	Checking
Boolean test of assignment	0	0.30	0	0.30	Checking
Possibly negative subscription	0	0.25	0.05	0.30	Checking
Constant value Boolean	0	0	0.25	0.25	Checking
Boolean within 'String' always evaluates to [True/False]	0	0.10	0.10	0.20	Checking
Referencing data from already freed pointer	0.20	0	0	0.20	Assignment
Logic error and Typo	0.05	0.10	0	0.15	Checking
Possible division by zero	0	0.15	0	0.15	Checking
Non-negative quantity is never less than zero	0	0	0.10	0.10	Checking
NULL pointer dereference	0.05	0.05	0	0.10	Assignment
Variable depends on order of evaluation	0	0	0.10	0.10	Checking
Dereferencing a constant string to a pointer	0.05	0	0	0.05	Assignment
Resources not freed	0	0.05	0	0.05	Assignment
Unrecognized format	0	0	0.05	0.05	Checking
Wrong output message	0	0	0.05	0.05	Checking
Total	19.57	38.81	41.62	100	

more than 800 bugs, but only 33 were found in students' projects. The faults were given one of the following severity levels, based on potential failure:

- **Critical** – this fault can cause application dump, service outage, system reboot;
- **Major** – this fault can cause segmentation fault, memory leaks, resource leaks, data corruption;
- **Minor** – this fault may result in unexpected behavior;
- **Coding standard** – code that violates coding standard and reduced readability and maintainability of the project.

The results are consistent with the 80-20 rule/ Pareto Principle, i.e. a great majority of the faults identified by few key programmer errors, as shown in table 9. "Possible use of NULL pointer" is most frequently error, identified by ASA – approximately 47% of all faults. About 92% of faults are focused on 10 fault types. To improve the code quality we will used this information in future educations to point the students what kind of programmer errors are most often happen in their projects.

There are some additional limitations on this research. First ASA tool outputs are screening. Second, assigning of severity level is a manually operation and is subjective.

4.5. Identification of Security Vulnerabilities

The question discussed in this section is: "*Can ASA help us to trap programming errors that have potential to cause security vulnerable?*"

To answer to it we used the followed metric:

- quantity of ASA faults by defect type.

We used static analysis tools for security vulnerable checking according [13, 14]. We highlighted the programmer's error, found by ASA that can potentially cause vulnerable. The results are present in table 10. These types of programmer errors have been documented by [15] as attacks. ASA tools have no context information and may produce wrong results sometimes.

The results indicate that ASA can be used to find security vulnerable errors.

5. Conclusions

To examine the quality of automated static analysis tools, we inspect three ASA tools. In this research we gather information about ASA tools fault detection, manually inspection faults and system testing failures in students' projects. Our analysis provides some results that are shown in Section 4. Using the received results we can conclude:

- The defect removal yield of ASA isn't significantly different from that of inspections. The defect removal yield of execution-based testing is two times higher than that of ASA and therefore may be more effective at finding the defects.

Table 9. Pareto Effect in ASA Faults

	% all faults	% critical faults	% major faults	% minor faults
Top 1 fault: Possible use of NULL pointer	46.89	62.15	39.06	43.62
Top 5 faults: Possible use of NULL pointer Possible access Out-Of-Bounds Pointer not freed or returned Memory leak Variable not initialized before using	75.13	87.27	57.36	77.14
Top 10 faults: Possible use of NULL pointer Possible access Out-Of-Bounds Pointer not freed or returned Memory leak Variable not initialized before using Inappropriate deallocation Suspicious use of Data overrun Type mismatch with switch expression Control flows into case/default	91.27	90.28	89.93	91.03

Table 10. Security vulnerable, detected by ASA tools

Fault Description	Explanation
Possible use of NULL pointer	Possibly cause the application to crash
Possible access Out-Of-Bounds	Perhaps on a buffer overflow attack Possibly cause denial of services
Suspicious use of	Malformed data Cross site scripting vulnerability
Type mismatch with switch expression	Possibly causing the application to crash if a user gives a float instead of a Boolean
Possibly passing a NULL pointer to function	Possibly cause application to crash
Passing NULL pointer to function	Possibly cause application to crash
Possibly division by zero	Possibly cause the application to crash Possibly cause denial of services
NULL pointer dereference	Possibly cause application to crash
Unrecognized format	Format string vulnerable, malformed data
Wrong output message	Cross site scripting vulnerability that can print out information about a system

- The ASA tools are effective for identifying two ODC types: Assignment and Checking.
- The inspection identifies Algorithm, Documentation and Checking faults.
- The great majority of the faults identified by few key programmer errors.
- "Possible use of NULL pointer" is most often fault, identified by ASA – approximately 47% of all faults.
- About 92% of faults are focused on 10 fault types.
- The ASA tools can be used to find security vulnerable errors.

In conclusion results indicate that ASA tools are economical complement to other testing techniques.

6. References

1. http://www.gimpel.com/html/products.htm
2. http://www.reasoning.com
3. http://klocwork.com
4. Basili, V.R., Green, S., et al.:*The Empirical Investigation of Perspective-Based Reading*. Empirical Software Engineering, Vol. 1, No. 2, 1996
5. Young, M., Taylor, R.N.: *Rethinking the Taxonomy of Fault Detection Techniques*. Proc. Conf. Software Eng., p. 53-62, 1989
6. Osterweil: *Integrating the Testing, Analysis, and Debugging of Programs*. Proc. Symp. Software Validation, 1984
7. Rutar, N., Almazan, C.B., Foster, J.S.: *A Comparison of Bug Finding Tools for Java*. Proc. IEEE Symp. Software Reliability Eng. (ISSRE), p. 245-256, 2004
8. Chillarege, R., Bhandari, I.S. et al.: *Orthogonal Defect Classification - A Concept for In-Process Measurements*. IEEE Trans. Software Eng., vol. 18, no. 11, p. 943-956, Nov. 1992
9. * * *: *IEEE Standard Classification for Software Anomalies*. IEEE Standard 1044-1993, 1993
10. Humphrey, W.S.: *A Discipline for Software Engineering*. Addison Wesley, 1995
11. Jones, C.: *Software Defect Removal Efficiency"*. Computer, Vol. 29, no. 4, p. 94-95, Apr. 1996
12. Jones, C.: *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley, May, 2000
13. Chess, B.: *Improving Computer Security Using Extended Static Checking*. Proc. IEEE Symp. Security and Privacy, p. 160-173, 2002
14. Chess, B., McGraw, G.: *Static Analysis for Security*. IEEE Security & Privacy, Vol. 2, no. 6, p. 76-79, 2004
15. http://www.securityfocus.com
16. Zheng, J., Williams, L. et al.: *On the Value of Static Analysis for Fault Detection in Software*. IEEE Trans. Software Eng., Vol. 32, no. 4, p. 240-253, April, 2006

Received in March 2007, and revised form in June 2007