Analysis Tool Evaluation:
GrammaTech CodeSonar

Final Report

April 24, 2007

Team
Vishal Garg
Sean Lao
Xiang Shen
Guo-Shiuan Wang
Bradley Wilson
Pengfei Zhao

Carnegie Mellon University

Document Revision History

Version	Date	Author(s)	Comments
0.1	4/22/07	Sean Lao	Initial Template
0.2	4/22/07	Xiang Shen	2.1; part of 2.2; 3.3; 5
0.3	4/22/07	Pengfei Zhao	Added section 2.1, updated 2.3, updated 3.1, 3.3 , appendix B. Added more columns of some table.(all changes are tracked)
0.4	4/23/07	Vishal Garg	Updated Section 2.3, 3.2, Appendix 6
0.5	4/23/07	Bradley Wilson	Added sections 1.1 and 1.2. Changed Section 2.2 to times new roman/12 to match other sections Added summary updates for Sockets app
0.6	4/23/07	Guo-shiuan Wang	Updated section 2.1, 2.3, 3.4, 3.5, 4 and Appendix 6
0.7	4/23/07	Xiang Shen	Added section 2.1; Updated 2.2
0.8	4/24/07	Bradley Wilson	Changed MB to KB in table. Added data to 3.2 and appendix B for sockets
0.81	4/24/07	Bradley Wilson	Fixed table 3.2
0.82	4/24/07	Pengfei Zhao	Modified experimentation summary and performance part
1.0	4/25/07	Sean Lao	Revision of entire document for grammar, formatting, and completeness

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Description

Static analysis tools provide another layer of protection when developing software, and when used correctly they can help improve the overall quality of the system. At a fundamental level, these tools can ferret out many bugs that will occur during run-time without actually running the program with volumes of test cases.

In an effort to explore how well one of these tools works, our team has conducted an evaluation of the static analysis tool CodeSonar, produced by GrammaTech. The tool is the flagship application produced by GrammaTech for analysis of C/C++ programs. Much of their research is government funded, but they also boast a reputable client list.

Although it can uncover a wide array of problems, some of the more interesting bugs it can uncover are buffer overrun and underrun, null pointer dereference, memory leaks, unreachable code, and concurrency locking issues.

You should glean from this document a basic understanding of the tool, as well as some critical analysis of its usefulness within the software engineering process. Other areas of consideration are general benefits and drawbacks, usability, and depth of information about the bugs found as it pertains to refactoring a potential software project.

1.2 How CodeSonar Works

CodeSonar operates as a listening application looking for programs that might invoke a C/C++ compiler. The tool runs concurrently with your C/C++ compilation tool such as MS Visual Studio, but it also functions in other environments such as Unix. Our evaluation version is for the Windows platform only, and the user must specify a type of compiler to look for before running the program.

The tool has a rather extensive list of configuration options, but the interface and documentation regarding them is quite limited. In fact, it is very similar to a verbose Linux configuration file. Much of the configuration options are for manipulating the time CodeSonar spends doing various analysis tasks, thus affecting the overall performance of the analysis. You can search for a specific bug by suppressing other types, or modify how the tool creates the abstract representation of your code.

Once you build your application, CodeSonar intercepts the code, and creates an abstract representation of the program. The nature of the eavesdropping is minimally invasive, and is one of the architectural features of the application. You do not have to alter your C/C++ compiler to use the tool. When the compilation is complete, the tool uses the files it created in the build process to synthesize a model for the program, and then executes that model to conduct the analysis.

The analysis can take some time depending on the size and complexity of the application you are analyzing. A small program will take no more than a minute, but large

applications can take hours, especially in default mode that looks for all types of bugs it is capable of finding. Once the analysis is complete, the results are shunted to a highly interactive HTML report, showing the number and types of bugs found.

Project Details	
Project:	csProject2
User:	bradleyw
Date:	Mon Apr 23 14:41:58 2007
Machine:	WILSON2-BRADLEY
Machine Type:	i686-pc-mingw32
Parse Errors:	0
Parse Warnings:	0

CodeSonar Results Summary		Visible With Unique Bug Points
Ignored Return Value	2	1
Leak	2	1
Total	4	2

CodeSonar Metrics Summary	
Total Lines	3978
Lines With Code	2775
Code Only Lines	2458
Comment Lines	654
Mixed Lines	317
Blank Lines	549

Figure 1: Example Summary HTML Page

A nice feature is that the code syntax is ported to HTML so that when you click on a bug in the HTML report, the syntax is opened, and the bugs are highlighted in yellow for easy viewing.

```
123          filePoint = OpenFile ( argv[2] );      /* Leak inside call (ID: 1) */
```

Figure 2: Example Bug Comment

Further investigation of the bug opens a subsequent screen that shows in a basic way how the abstract representation came to this conclusion.

```
allocated inside call     123 +        filePoint = OpenFile ( argv[2] );      /* Leak (ID: 1) */
referenced by $ heap_41   129          if (!filePoint)
leaked                    140              return -1;
```

Figure 3: Example Bug Rationale

The following image is a graphical representation of the process.

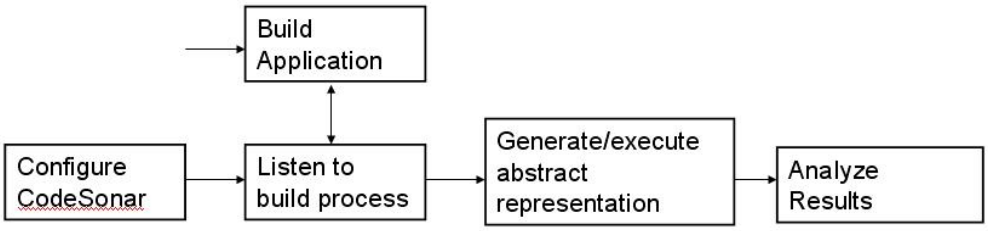

Figure 4: CodeSonar Execution Process

2 Experimentation

2.1 Purpose

We applied CodeSonar to different types of software on different compilation tools. Thus we were able to get a comprehensive understanding and evaluation about CodeSonar. Altogether, we ran CodeSonar against six different applications.

2.2 Test Bench

The experiments are classified into the following groups:

- ANSI C platform independent application
  - Small size: Messaging
- Windows applications built by Microsoft Visual Studio 2005 under Windows XP.
  - Medium size: Scribble (Visual Studio 2005 sample)
  - Large size: WordPad (Visual Studio 2005 sample) and FileZilla
  - Huge size: eMule 0.44a
- Non-Windows applications built by GCC under Cygwin.
  - Medium size: Sockets

These experiments range from small to large sizes, platform-independent to platform-dependent, C to C++, and buggy to well-tested software. This enabled us to observe the combined effects of multiple factors on CodeSonar.

2.3 Instructions

2.3.1 Installation

Installation of CodeSonar is described by the following steps:

Windows:

1) Run the installer, following the on-screen prompts to install the software.
2) Save the license file to disk, then set the environment variable LM_LICENSE_FILE to point to it.

UNIX:

1) Untar the tarball to the directory of your choice. The contents of the tarball will be extracted into ./CodeSonar-2.1p0.
2) Add codesonar/bin to PATH environment variable for the user.

2.3.2 Execution

Execution of CodeSonar is described by the following steps:

Windows GUI:

1) Start up your regular build environment (for example, Visual C++) and load the software project that is ready for a build.
2) Invoke the CodeSonar build wizard from the Windows Start menu:
3) On the first screen of the wizard specify a Save As name and directory for your project and click Next.

4) On the second screen of the wizard, click Record.
5) Build your software project in your regular build environment.
6) On the third screen of the wizard, click Finalize.
7) On the fourth screen of the wizard, click Browse Bugs.

Command Line:
1) In the command line, execute:
   ```
   codesonar hook-html <project-name> <command>
   ```

   where:
   project-name is the name you want to use for your CodeSonar project, and
   command is the command you usually use to build your project on the command line (e.g., make TargetProject).
2) Browse *dir/project-name.html* to check bugs.

2.4 Summary

Table 1 describes the summary of our experiment results.

Table 1: Summary of Experiment Results

Application	Lines of Code	Total Defects	Number of Software Defects	Number of Development environment Defects	True Defects/ KLOC	Total Defects/ KLOC
Messaging	941	4	4	0	4.25	4.25
Scribble	44188	24	2	22	0.045	0.543
Sockets	49274	16	14	2	0.284	0.325
FileZilla	178900	701	607	94	3.393	3.918
WordPad	193182	109	77	32	0.399	0.564
eMule(0.47a)	504242	1126	604	522	1.198	2.233
AVERAGE	*161788*	*330*	*218*	*112*	*1.347*	*2.040*

The table lists the analysis result of CodeSonar for six applications ranging from small to large sizes. The total defects found in these applications varied from less than 10 to more than 1000 when the software size increased. During the analysis, we found that a large amount of defects were located in code automatically generated by the development environment. We will consider the defects found in the software itself as “true” software defects. Based on our evaluation, about two-thirds of defects found by CodeSonar are true software defects. Appendix B contains the details for the specific bug types found for each application

3 Analysis

3.1 Usability

CodeSonar is a source-code analyzer that identifies complex bugs at compile time. It supports sophisticated C/C++/Ada source code analysis on major platforms, including Linux, Windows and Solaris. The tool also supports most modern compilers such as GCC, G++, Microsoft Visual Studio, Sun CC. Due to the limitations of the evaluation version

of CodeSonar we obtained, we have only evaluated it on the Windows platform with Microsoft Visual Studio 2005 and Cygwin GCC.

The main GUI of CodeSonar on the Windows platform is not intuitive, so inexperienced users may have trouble using the tool for the first time. Fortunately, CodeSonar provides a clear user manual with detailed information about operation steps and explanations about analysis results. Figure 5 contains a snapshot of the CodeSonar GUI in Windows:

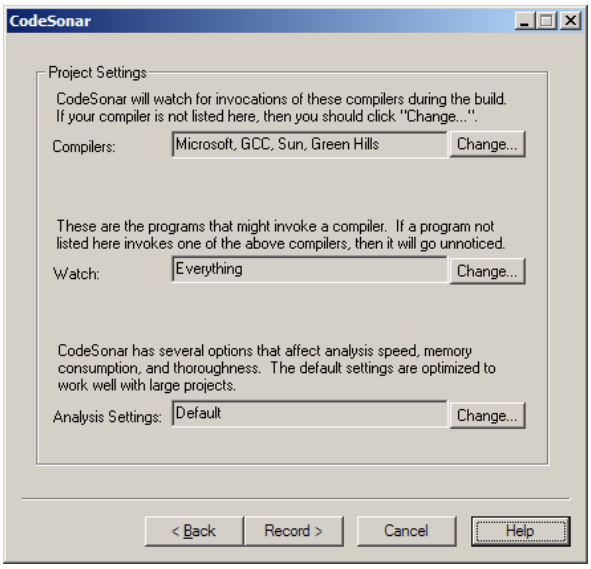

Figure 5: CodeSonar GUI

The screenshot in Figure 6 is captured when CodeSonar is recording compilation information:

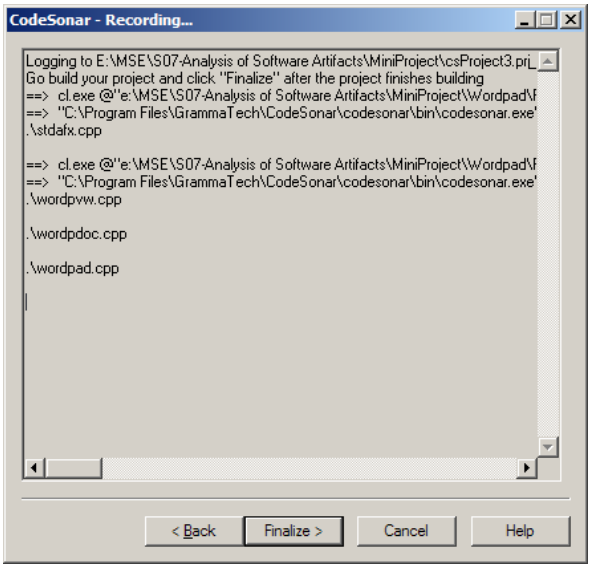

Figure 6: CodeSonar Recording Compilation Information

There are many usability issues with this non-intuitive GUI. First, CodeSonar does not provide a progress bar during the analysis, so users cannot determine how long the

analysis may take. Second, the functionality of the "Finalize" button is unclear. According to the user manual, this button should become enabled only after all analysis has finished. But in our experiments, this button may become enabled during the analysis. If the user clicks this button before analysis is complete, we can only get an incomplete report. Finally, the "Help" button here does not work.

After CodeSonar finishes its analysis, a cleanly organized report will be generated. In this report, we can easily inspect, filter, and check the analysis results. Figure 7 is an example defect report for a "Missing Return Statement".

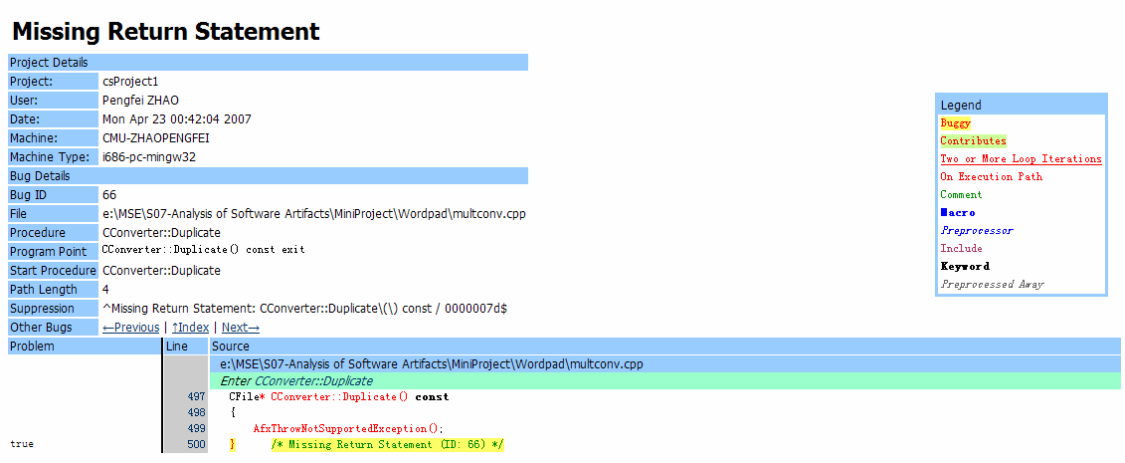

Figure 7: "Missing Return Statement" Defect Report

As a user, you can easily jump between different parts of the code base by clicking on function/variable names or bug descriptions. An extremely useful feature of this report is the detailed call-chain of each function call. For example, the screenshot in Figure 8 captures how a possible memory leak defect can be viewed in this report. When a user clicks the "+" symbol in front of a function call, CodeSonar will expand that function to expose the function body code, with call chain descriptions appearing to the left of the code. This is a really useful feature for a developer to analyze the root cause of this memory leak defect. Furthermore, the tool also provides the pre-conditions and post-conditions related to that specific bug, which are also extremely helpful.

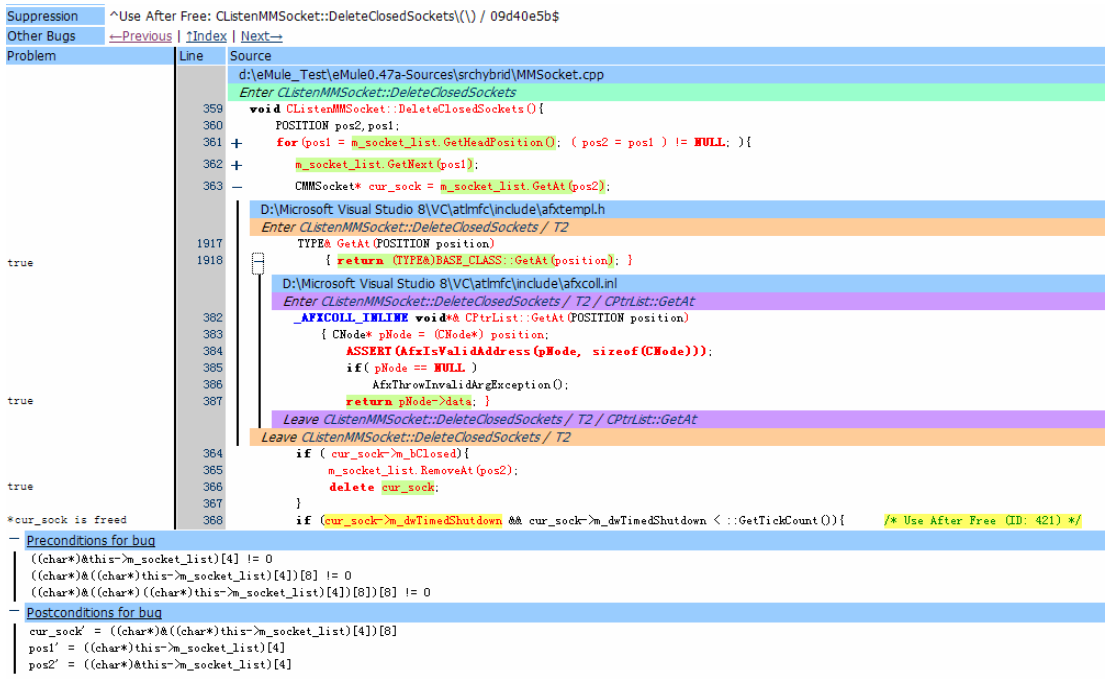

Figure 8: Call Chain View for a Bug

Generally, although there are many usability issues of the main GUI of CodeSonar, we were pleasantly surprised by the analysis report. Despite having some disadvantages, CodeSonar is still a great static source code analysis tool for developers to pinpoint why a particular bug occurs.

3.2 Performance

We evaluated the performance of CodeSonar based on the criteria in Table 2. We found that, on average, CodeSonar can find 1.75 real software defects per minute. On our test applications, the hard disk space required to store the analysis data files ranged from 20MB of space to an enormous 5GB of space. The time spent analyzing each application ranged from under a minute to 5.5 hours. Generally, the analysis takes longer as the size of the application increases. The number of defects found also affects the total time required for analysis. We performed the tests with the default analysis settings (claimed by CodeSonar to work well with large projects). The performance may have improved if we had specified minimal status messages and targeted specific bug types.

Table 2: Experiment Performance Data

Application	Lines of Code	Hard Disk Space Required for Results (MB)	Time Spent (mins.)	Number of Software Defects	Number of Defects per Min.
Messaging	941	20	0.5	4	8
Scribble	44,188	600	18	2	0.11
Sockets	49,274	109	8	14	1.75
FileZilla	178,900	2048	240	607	2.53
WordPad	193,182	657	10	77	7.7
eMule (0.44a)	504,242	5250	330	604	1.83
AVERAGE	*161788*	*1447*	*101*	*218*	*1.75*

3.3 Limitations

Since CodeSonar analysis is performed without actually executing programs built from the target software, only static code analysis is performed. Defects which could be found easily by dynamic code analysis may not be well identified by this tool. Performance analysis such as profiling is not available due to the static nature of the tool. Functionality defects are also in the blind spots of CodeSonar because dynamic analysis is not supported.

CodeSonar does not account for the system environment, which it abstracts away. The tool depends on the areas that a test scaffold covers. The types of bugs that are caused by inputs not handled by the scaffold, particularly hardware-related issues, survive unnoticed.

3.4 Types of Defects Found

This section describes the types of defects that we found in our experimentation with CodeSonar.

3.4.1 Fatal and Critical Defects

According to our experimentation results, some bugs that may crash the system were found. Some of these we describe as fatal, which means they will definitely crash the system if encountered. Fatal errors include the “divide by zero” and “null pointer dereference” bugs. Others bugs may or may not crash the system depending on the runtime behavior of the system and the compiler. The following list contains the fatal and critical errors we found. Refer to Appendix A for descriptions of these bugs.

- Fatal Errors: Definitely crashes the system
    - Divide by zero
    - Null pointer dereference
    - Double free
    - Free null pointer
    - Malloc / Memcpy buffer length unreasonable
    - Use after free
- Critical Errors: May crash the sysetm
    - Buffer overrun / underrun
    - Dangerous function Cast
    - Format string
    - Ignored return value
    - Leak
    - Negative file descriptor
    - Missing return statement
    - Null test after deference
    - Type underrun
    - Redundant condition
    - Uninitialized variable
    - Delete objects created by new[]

3.4.2 Innocuous Defects

Based on the experiment results, some innocuous defects were also found. Innocuous defects are defects which do not affect the functionality of the system, but are rather elements of bad coding style which may cause future defects. All of the errors we found show logical errors committed by the developers. The following list contains the innocuous errors we found.

- Useless Assignment
- Unreachable code
- Unused value

3.4.3 Implications

Our experimentation data in Appendix B show that CodeSonar found significant amounts of both fatal/critical defects and innocuous defects. The fatal defect which was found the most times was “null pointer dereference,” which had 109 occurrences in eMule and 73 occurrences in FileZilla. The innocuous defect which was found the most was “uninitialized variable,” which had 114 occurrences in eMule and 82 occurrences in FileZilla. Based on a random sample set of bugs that we analyzed, we did not find any bugs that were false positives.

4 Conclusions and Recommendations

CodeSonar is an excellent tool for finding possible bugs in the software. The reported information is helpful for the developer to fix fatal and critical bugs and to think of some other potential bugs in the software system. The organization of the analysis results allows clear views of both summary data and detailed data.

The drawbacks that we are really concerned about are its performance and system resource consumption. It takes a long time to finish the analysis on very large code bases (magnitude of hundreds of KLOC). For these code bases, the analysis also generates huge amounts of data which occupies extremely large chunks of disk space. If your system is large-scale, we suggest you to run the tool using a fast machine with a large amount of memory and disk space (see our experimentation data for possible resources usage). Adjusting the configuration of message detail and reported bug types can also improve the performance of CodeSonar. One other drawback which does not concern us as much is the lack of descriptions in the user interface. This type of problem is annoying for inexperienced users and should not occur in commercial software.

CodeSonar can be used in all testing phases of the software development lifecycle. We recommend that CodeSonar be configured to report the minimal status messages and only include fatal and critical errors for unit testing. During this phase, fatal and critical errors are more likely to be encountered since the code has not yet been tested much. Also, the developer is most likely to understand fully the logic of his or her code during unit testing, so more detailed status messages in CodeSonar are unnecessary.

For integration and system testing, we recommend that all types of bugs be reported by CodeSonar with normal or maximum message detail. During this stage, there should be more bugs which are less critical since the most critical bugs were checked during unit testing. Running CodeSonar on a large code base with most features turned on may require a huge amount of time so it may save time to run CodeSonar on different machines for independent modules.

5 Appendix A – Defect Descriptions

The following are descriptions of all the bugs that CodeSonar can find. These descriptions were taken from [1].

- **Buffer Overrun**: A read or write to data after the end of a buffer.
- **Buffer Underrun**: A read or write to data before the beginning of a buffer.
- **Type Overrun**: An overrun of a boundary within an aggregate type.
- **Type Underrun**: An underrun of a boundary within an aggregate type.
- **Null Pointer Dereference**: An attempt to dereference a pointer to the address 0.
- **Divide By Zero**: An attempt to perform integer division where the denominator is 0.
- **Double Free**: Two calls to free on the same object.
- **Use After Free**: A dereference of a pointer to a freed object.
- **Free Non-Heap Variable**: An attempt to free an object which was not allocated on the heap, such as a stack-allocated variable.
- **Uninitialized Variable**: An attempt to use the value of a variable that has not been initialized.
- **Leak**: Dynamically allocated storage has not been freed.
- **Dangerous Function Cast**: A function pointer is cast to another function pointer having an incompatible signature or return type.
- **Delete[] Object Created by malloc**: An attempt to release memory obtained with malloc using delete[]
- **Delete[] Object Created by new**: An attempt to release memory obtained with new using delete[]
- **Delete Object Created by malloc**: An attempt to release memory obtained with malloc using delete
- **Delete Object Created by new[]**: An attempt to release memory obtained with new[] using delete
- **Free Object Created by new[]**: An attempt to release memory obtained with new[] using free
- **Free Object Created by new**: An attempt to release memory obtained with new using free
- **Missing Return Statement**: At least one path through a non-void return-type function does not contain a return statement.
- **Redundant Condition**: Some condition is either always or never satisfied.
- **Return Pointer To Local**: A procedure returns a pointer to one of its local variables.
- **Return Pointer To Freed**: A procedure returns a pointer to memory that has already been freed.
- **Unused Value**: A variable is assigned a value, but that value is never subsequently used on any execution path.
- **Useless Assignment**: Some assignment always assigns the value that the variable being modified already has.
- **Varargs Function Cast**: A varargs function pointer is cast to another function pointer having different parameters or return type.
- **Ignored Return Value**: The value returned by some function has not been used.

- **Free Null Pointer**: An attempt to free a null pointer.
- **Unreachable Code**: Some of the code in a function is unreachable from the function entry point under any circumstances.
- **Null Test After Dereference**: A pointer is NULL-checked when it has already been dereferenced.
- **Format String**: A function that should have a format string passed in a particular argument position has been passed a string that either is not a format string or is from an untrusted source. (Potential security vulnerability.)
- **Double Close**: An attempt to close a file descriptor or file pointer twice.
- **TOCTTOU Vulnerability**: A time-of-check-to-time-of-use race condition that can create a security vulnerability.
- **Double Lock**: An attempt to lock a mutex twice.
- **Double Unlock**: An attempt to unlock a mutex twice.
- **Try-lock that will never succeed**: An attempt to lock a mutex that cannot possibly succeed.
- **Misuse of Memory Allocation**: Incorrect use of memory allocators.
- **Misuse of Memory Copying**: Incorrect use of copying functions.
- **Misuse of Libraries**: Misuse of standard library functions.
- **User-Defined Bug Classes**: Checks for arbitrary bug classes can be implemented through the CodeSonar extension functions.

6 Appendix B – Experimentation Data

This section provides the experimentation data of our test applications. For each bug type, the number of true defects found is listed.

6.1 Scribble

Bug Type	Number Found (True defects)
Useless Assignment	2

6.2 WordPad

Bug Type	Number Found (True defects)
Buffer Underrun	3
Dangerous Function Cast	4
Missing Return Statement	1
Null Pointer Dereference	3
Redundant Condition	7
Uninitialized Variable	36
Unreachable Code	5
Unused Value	14
Useless Assignment	4

6.3 eMule

Bug Type	Number Found (True defects)
Buffer Overrun	15

Bug Type	Number Found (True defects)
Buffer Underrun	4
Dangerous Function Cast	40
Division By Zero	5
Free Null Pointer	6
Ignored Return Value	31
Leak	27
Negative file descriptor	9
Null Pointer Dereference	109
Null Test After Dereference	11
Redundant Condition	85
Uninitialized Variable	114
Unreachable Code	69
Unused Value	29
Use After Free	2
Useless Assignment	43
delete Object Created by new[]	4
malloc Buffer Length Unreasonable	1

6.4 Sockets

Bug Type	Number Found (True defects)
Null Pointer Dereference	2
Redundant Condition	1
Uninitialized Variable	3
Unreachable Code	4
Unused Value	3
Ignored Return Value	1

6.5 Messaging

Bug Type	**Number Found (True defects)**
Leak	2
Ignored Return Value	2

6.6 FileZilla

Bug Type	**Number Found (True defects)**
Buffer Overrun	21
Buffer Underrun	4
Dangerous Function Cast	109
Double Free	1

Bug Type	Number Found (True defects)
Format String	1
Free Null Pointer	3
Ignored Return Value	25
Leak	4
Negative file descriptor	4
Null Pointer Dereference	73
Null Test After Dereference	3
Redundant Condition	36
Type Underrun	10
Uninitialized Variable	82
Unreachable Code	84
Unused Value	122
Useless Assignment	23
memcpy Length Unreasonable	2

7 References

[1] GrammaTech Representative List of CodeSonar Checks, http://www.grammatech.com/products/codesonar/listofchecks.html, last viewed on 4/26/07.