

The background of the page is a complex, abstract composition of overlapping, semi-transparent rectangular and circular shapes in various shades of gray and black, creating a sense of depth and movement.

How Open-Source and Commercial Software Compare:

**A Quantitative Analysis
of TCP/IP Implementations
in Commercial Software
and in the Linux Kernel**

**Executive
Summary**

Proponents of Open Source software have long claimed that their code is of higher quality than the equivalent commercial software. Opponents of Open Source argue just the opposite: that Open Source software is inherently unreliable. There is little independent, objective data available to support either view. Reasoning's automated inspection service makes possible, for the first time, meaningful comparisons of different software projects. Using static analysis technology, Reasoning identifies critical defects at the source code level—independent of requirements specifications or test suites.

Reasoning undertook a study to identify quantitative differences in Open Source and commercial software. Comparing different implementations of the same function provides more insight into any differences found. Reasoning compared six different implementations of the TCP/IP protocol, the fundamental protocol that underlies the Internet. We chose TCP/IP because the functional requirements are well defined and stable; the implementation is non-trivial; and it is a critical component of every computer system and many embedded devices.

Five implementations of TCP/IP from a variety of commercial systems, both general purpose operating systems and embedded applications, were inspected. Reasoning also inspected the TCP/IP code in version 2.4.19 of the Linux kernel. This document summarizes the results of that inspection and compares the results with the results from the inspections of the commercial implementations.

Based solely on quantitative criteria used in all of Reasoning's inspection projects, the Open Source implementation of TCP/IP in the Linux operating system exhibited a defect density of 0.10 defects per thousand lines of source code (Defects/KLSC), while the composite defect density of the five commercial projects was 0.55.

In its most recent analysis of 200 commercial projects totaling 35 million lines of source code, Reasoning determined that 33 percent had defect densities below 0.36; 33 percent had defect densities between 0.36 and 0.71, and the remaining 33 percent had defect densities above 0.71. Thus, the TCP/IP implementation in the Linux operating system ranks in the upper third, while the composite code quality of the commercial implementations ranks in the middle third.

Table of Contents

Executive Summary	1
Table of Contents	2
Introduction	3
What is Open Source, and Why Might it Be Better?	3
Automated Software Inspection	4
Inspection Complements Testing	5
Reasoning’s Automated Software Inspection Service	5
Detailed Defect Reporting	6
High-Risk Files	7
Industry Defect Comparison	7
Study Methodology	7
The Commercial Projects	8
The Linux Inspection	9
Feedback from the Developers	9
Example Defect Description	10
Results	11
Defect Repair	11
Conclusions	12
About Reasoning	12

Introduction

Reasoning Inc. provides an automated software inspection service that is used by leading commercial software vendors to identify defects and provide metrics regarding the quality of the inspected code. Reasoning's inspection service is based on a combination of technology and a repeatable process, and enables Reasoning to maintain a database of metadata about code quality. This database provides a unique opportunity to independently assess the quality of software.

This document builds towards its conclusion through the following flow:

- Why Open Source is claimed to exhibit higher code quality
- Why Linux was chosen for this study
- How automated software inspection works
- The results of the Linux inspection
- The results of the commercial inspections
- The results compared
- The conclusion

What is Open Source, and Why Might it Be Better?

Most commercial software vendors distribute their products in the form of executable or object code. Their customers do not acquire a license to use the source code, so they cannot change or extend the functionality of the executables, except by specific arrangements with the vendor. They are generally prohibited from redistributing a changed or extended version to others. With few exceptions, customers of commercial software vendors must rely on the vendor to make changes and extensions.

Open Source software represents a fundamentally different way in which software is developed, sold, and maintained. For a precise definition of the term, refer to www.opensource.org/docs/definition_plain.html. Below are the three characteristics that are most relevant to this study:

- The source code of Open Source programs is accessible to users such that they can make changes or extensions to that code.
- Changes and extensions are freely redistributable.
- The source code can be modified by many people without the need for those people to be employed by the same software vendor.

Open Source proponents believe that, for important pieces of software, the Open Source model encourages several activities that are not common in the development of commercial code:

- Many users don't just report bugs, as they would do with commercial software, but actually track down their root causes and fix them.
- Many developers are reviewing each other's code, if only because it is important to understand code before it can be changed or extended. It has long been known that peer review is the most effective way to find defects.
- The Open Source model encourages programmers to organize themselves around a project based on their contributions. The most effective programmers write the most crucial code, review the contributions of others, and decide which of these contributions are incorporated into the next release.
- Open Source projects don't face the same type of resource and time pressures that commercial projects do. Open Source projects are rarely developed against a fixed timeline, affording more opportunity for peer review, and usually offer extensive beta testing before "release."

For these reasons, Open Source enthusiasts claim that the Open Source model produces better quality software than commercial software development.

Automated Software Inspection

Software inspection—the process of examining source code to identify defects—is a standard practice in development organizations and is widely recognized as the best way to find defects. Inspection is hardware-independent, does not require a "runable" application nor a suite of test cases, and does not affect code size or execution speed. But until recently, it has been a manual process—very slow, and very costly—or tools-based and hard to implement effectively.

The majority of code inspections are performed manually. Although a human reading the code line-by-line can theoretically uncover the greatest number of defects, the process is slow, painstaking, and fraught with inconsistency. Also, this approach does not scale to handle today's multi-million line applications. As a code base grows, the cost of a complete manual inspection becomes prohibitive and the volume of code is intimidating to developers. In practice, manual inspections are only performed on subsets of the source code.

Inspection tools are able to perform only a portion of the inspection process, requiring significant further manual review. The inspection tools generate a large volume of defect "warning messages," many of which are, in fact, false positives. The inspection tool "thinks" it has found a defect, but a deeper manual analysis of the context shows that the reported issue is not actually a defect. This false positive problem is very severe. Frequently, the rate will exceed 50 false positives to each true positive; in other words, only two percent of the warning messages represent defects.

Inspection Complements Testing

One of the best ways to improve the overall quality of software is to improve the effectiveness of software testing. Testing is acknowledged to be a critical part of total quality assurance, but it has limitations, including:

- **Cost:** It is expensive and time-consuming to create, run, validate, and maintain test cases and processes.
- **Code coverage:** Typically, only a percentage of statements are tested. With large, complex applications, often less than 40 percent of the code is covered by test cases.
- **Tracking down defects:** It can be difficult and time-consuming to trace a failure from a test case back to the root cause so that developers know what code to change.

So, while comprehensive software testing is a vital part of any quality assurance program, it is not a panacea. Software testing, although extremely valuable, is inadequate in light of the increasing need for high reliability in very large code bases. Testing alone cannot guarantee defect-free code, nor can it ensure a sufficiently high level of software quality. But when testing is combined with automated software inspection services, significant quality improvements can be achieved.

Reasoning's Automated Software Inspection Service

Reasoning's automated software inspection service provides many of the benefits of a manual code review in significantly less time and at dramatically lower cost than manual inspection or internal use of inspection tools. With Reasoning's service, in-house resources are not diverted from current development projects. The Reasoning service identifies defects that cause application crashes and data corruption, and provides actionable reports. For example, the error classes in C and C++ include:

- Memory leak
- NULL pointer dereference
- Bad deallocation
- Out of bounds array access
- Uninitialized variable

The results of a Reasoning inspection are reports that:

- Make defect analysis fast and simple by identifying the location and describing the circumstances under which the defects will occur
- Identify the parts of the code with the greatest risk, enabling the development organization to focus QA and testing resources where they are most needed
- Compare the customer's code quality with a benchmark

Detailed Defect Reporting

Below is a sample defect, as it would be reported in the Detailed Defect Report. For each defect, the report identifies the class of defect, its exact location by file and line number, a description of the specific defect, the preconditions that must hold for the defect to occur, and a fragment of the source code with key lines highlighted.

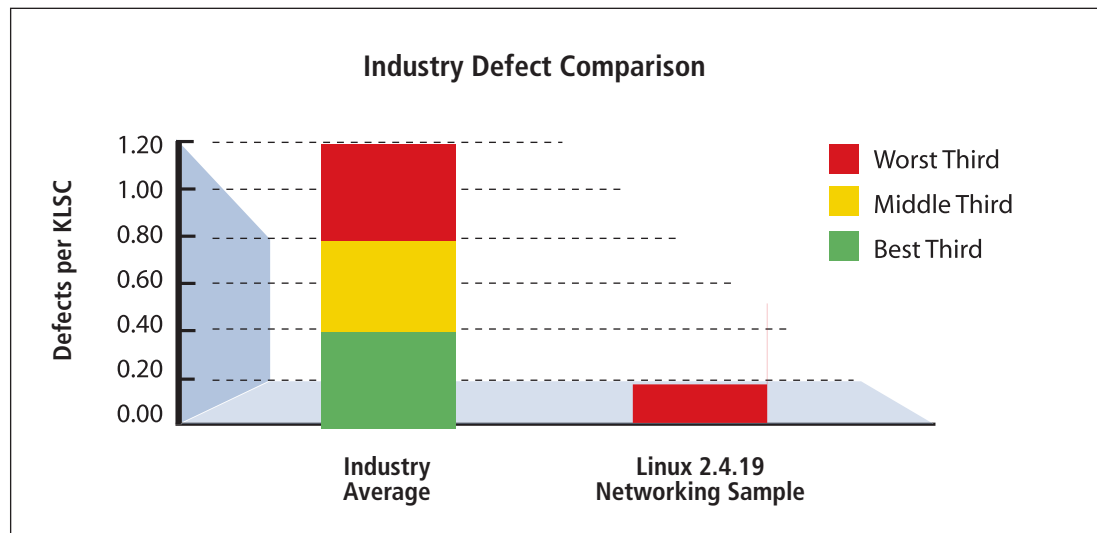
12/03/2002	Linux 2.4.19 Networking Sample from Reasoning Inc.
Defect Class	Null Pointer Dereference
Location	src/linux-2.4.19/net/sunrpc/xprt.c: 435
Description	The local pointer variable sock , declared on line 400 , and assigned on line 400 , may be NULL where it is dereferenced on line 435 .
Preconditions	The conditional expression (!xprt->addr.sin_port) on line 412 evaluates to false AND The conditional expression (!xprt_lock_write(xprt, task)) on line 417 evaluates to false AND The conditional expression (xprt_connected(xprt)) on line 419 evaluates to false AND The conditional expression sock on line 422 evaluates to false AND The conditional expression (!(inet = xprt->inet)) on line 425 evaluates to false .
Impact	A NULL pointer dereference usually causes a program exception. Notable exceptions to this rule are some embedded environments, in which a NULL pointer dereference does not cause program termination.
Code Fragment	<pre> 396 void 397 xprt_reconnect(struct rpc_task *task) 398 { ... 400 struct socket *sock = xprt->sock; ... 412 if (!xprt->addr.sin_port) { 413 task->tk_status = -EIO; 414 return; 415 } 416 417 if (!xprt_lock_write(xprt, task)) 418 return; 419 if (xprt_connected(xprt)) 420 goto out_write; 421 422 if (sock && sock->state != SS_UNCONNECTED) 423 xprt_close(xprt); 424 status = -ENOTCONN; 425 if (!(inet = xprt->inet)) { 426 /* Create an unconnected socket */ 427 if (!(sock = xprt_create_socket(xprt->prot, &xprt->timeout))) 428 goto defer; 429 xprt_bind_socket(xprt, sock); 430 inet = sock->sk; 431 } 432 433 /* Now connect it asynchronously. */ 434 dprintk("RPC: %4d connecting new socket\n", task->tk_pid); 435 status = sock->ops->connect(sock, (struct sockaddr *) &xprt->addr, 436 sizeof(xprt->addr), O_NONBLOCK); </pre>
Reasoning Inc.	Defect Data
	Page 11

High-Risk Files

The key metric reported for each project is defect density, expressed as defects per thousand lines of source code (Defects/KSLC). For example, if a 100,000-line application had 23 defects, its defect density would be .23. The analysis computes defect density file-by-file as well as for the entire project. From our metadata, we know that a small percentage of source files (< 10 percent) frequently have a defect density an order of magnitude higher than the rest of the application. A high-risk file report shows these high-risk source files, ranked from highest to lowest defect density. This information allows a development team to focus on the portion of the application requiring additional manual inspection, extra testing, or redesign.

Industry Defect Comparison

The overall project defect density is reported relative to a representative sample of inspections performed by Reasoning. In this aggregation of 200 projects totaling 35 million lines of code, 33 percent had a defect density below 0.36 Defects/KSLC, 33 percent had a defect density between 0.36 and 0.71 Defects/KSLC, and the remaining 33 percent had a defect density above 0.71 Defects/KSLC.



Study Methodology

Of the thousands of Open Source applications available, we chose the Linux® operating system. This general-purpose operating system has been under development for nearly a decade, is widely used and is actively maintained and enhanced by a community of thousands of programmers.

However, comparing the quality of several entire operating systems is a challenge, primarily because the size, scope and goals can be so different. Instead, we chose a common function implemented by all the projects in our study, the TCP/IP network protocol “stack.” There were several reasons for this decision:

- these protocols are precisely defined and documented in published standards;
- the most widely used versions (version 4) have been stable for several years;
- there is a large body of published literature on implementing the protocols;
- a complete implementation is non-trivial; and
- these protocols are usually in the operating system “kernel,” the lowest level software in the system; thus, defects can have a major impact, including inability to communicate, system crashes, network outages, and security violations.

Each project was inspected using the standard Reasoning automated software inspection process.

The Commercial Projects

Reasoning has conducted inspections of five different commercial TCP/IP implementations, including implementations from both general-purpose operating systems and embedded applications.

Four of the five implementations are considered mature, having been in commercial use for over ten years (although the TCP/IP code is under active development). The fifth is relatively young: it was started about three years ago.

The size of these projects ranges from 64 KLSC to 269 KLSC. The table below summarizes the information about the commercial projects.

Name	Size (KLSC)	Maturity (years)
A	64.3	>10
B	85.1	>10
C	269.1	>10
D	64.8	>10
E	84.7	3

For reasons of client confidentiality, we cannot disclose further information about these projects.

The Linux Inspection

We inspected the TCP/IP implementation in version 2.4.19 of the Linux kernel. We chose this version because it was the latest “stable” release at the time of the study. The TCP/IP code was inspected in isolation from the rest of the kernel, using the exact same process Reasoning uses for customer projects.

The Open Source TCP/IP implementation includes 166 source files with just under 82 thousand lines of source code (KSLC) in size, not including user include files, header files, blank lines and comments.

Reasoning found eight defects, resulting in a defect density of 0.10 defects/KSLC. The table below details, per defect class, how many defects were found in the application.

Inspection Class	Defect Instances
Memory Leak Reference to allocated memory is lost	1
NULL Pointer Dereference Expression dereferences a NULL pointer	3
Bad Deallocation Deallocation is inappropriate for type of data	0
Out of Bounds Array Access Expression accesses a value beyond the array	3
Uninitialized Variable Variable is not initialized prior to use	1
Total Defect Instances	8

Feedback From the Developers

We submitted the details to people on the kernel networking list and have received the following feedback so far:

- The memory leak is a real defect. Independently of this inspection, it has been fixed in version 2.4.20.
- The out of bounds array accesses are not real defects, because the kernel would not work if they were.
- The uninitialized variable is not a defect. This is code implementing a tiny interpreter, and the uninitialized variable represents variables in the interpreted language. These variables have random values when the interpretation starts, and it is the responsibility of the interpreted program to initialize the variables before they are used.
- We have not received definitive feedback on any of the null pointer dereferences.

In summary: one defect is real, four defects are not real, and three are undecided. A detailed description of one of the defects uncovered is presented below.

Example Defect Description

The accompanying defect data report shows the details necessary to assess each defect. As an example, here is the detailed description of the memory leak.

Defect Class:	Memory Leak
Location:	src\linux-2.4.19\net\socket.c : 750
Description	Local variable fna , declared on line 735 , is assigned a pointer to a block of memory allocated by kmalloc on line 741 . No other pointer refers to this memory block, so it is inaccessible (still allocated, but unreachable) once fna goes out of scope after line 750 .
Preconditions	The conditional expression (on) on line 739 evaluates to true AND The conditional expression (fna==NULL) on line 742 evaluates to false AND The conditional expression ((sk=sock->sk) == NULL) on line 749 evaluates to true .
Impact	Depending on how long the application runs, how frequently the leak occurs, and the amount of available (virtual) memory, memory leaks will sooner or later cause performance degradation of the application, and potentially of the entire system. Eventually, the performance degradation may lead to a fatal out-of-memory condition. This condition may be encountered by an application unrelated to the one that caused the memory leak.

Code Fragment

```

733 static int sock_fasync(int fd, struct file *filp, int on)
734 {
735     struct fasync_struct *fa, *fna=NULL, **prev;
736     struct socket *sock;
737     struct sock *sk;
738
739     if (on)
740     {
741         fna=(struct fasync_struct *)kmalloc(sizeof(struct fasync_struct), GFP_KERNEL);
742         if(fna==NULL)
743             return -ENOMEM;
744     }
745
746
747     sock = socki_lookup(filp->f_dentry->d_inode);
748
749     if ((sk=sock->sk) == NULL)
750         return -EINVAL;
751
752     lock_sock(sk);
753
754     prev=&(sock->fasync_list);
755
756     for (fa=*prev; fa!=NULL; prev=&fa->fa_next,fa=*prev)
757         if (fa->fa_file==filp)
758             break;
759

```

Results

The table below summarizes the results for all five inspection classes.

	Total Defects in Five Commercial Implementations	Total Defects in One Open Source Implementation
Memory Leak	43	1
Null Pointer Dereference	128	3
Bad Deallocation	0	0
Out of Bounds Array Access	9	3
Uninitialized Variable	132	1
Totals	312	8

Note that there are no *bad deallocations*. However, since the applications are generally fairly mature and all are written in C rather than C++, this is not particularly surprising. Bad deallocations that occur in C are generally beginner's mistakes (much more so than the other defect classes), and tend to happen on the path the code is intended to take, so there is a large likelihood they get caught quickly.

In light of the relative maturity of the commercial applications and the Open Source model underlying the Linux code, perhaps one should be surprised that any defects remain at all. Given the amount of testing that all the code bases had undergone before the inspections, this also confirms that testing is not enough: inspection finds defects that escape testing.

Of course, the five commercial applications together contain much more source code than the one Open Source application. Therefore it makes much more sense to look at defect densities (defects per KLSC) as shown in the table below.

Defect Repair

Since those most familiar with the application are best able to determine the need to repair any individual defect, the most reliable metric is which defects need to be fixed according to the developers or maintainers of the code.

The table below reflects the reported defects, the repaired defects, and the defect density (defects/KLSC) for the commercial projects and the Open Source project. Since we have not yet received feedback on many of the defects reported to the Linux kernel maintainers, the real number for the Open Source code may be higher.

	Defects		Size (KLSC)	Defect Densities (Defects/KLSC)	
	Reported	Repaired		Reported	Repaired
Commercial Implementations	312	235	568.0	0.55	0.41
Open Source	8	1	81.9	0.10	0.013

On average, both the reported and the repaired defect densities are higher for the commercial implementations compared to the Open Source implementation.

Conclusions

This study compares five commercial implementations of TCP/IP, the fundamental protocols underlying the Internet, with the TCP/IP implementation in version 2.4.19 of the Linux kernel, an Open Source general purpose operating system.

The Open Source implementation of TCP/IP in the Linux kernel exhibits significantly lower defect density when compared to the five commercial applications and falls within the “best third” of source code projects inspected by Reasoning.

About Reasoning

Reasoning is a leading provider of automated software inspection services that helps development organizations reduce the time and cost involved in finding software defects. The company’s business is focused on organizations that develop Java, C, and C++ applications.



For more information, contact:

Reasoning, LLC

PO Box 478

Menlo Park, CA 94026-0478

650 324-2510 (phone)

415 762-1992 (fax)

Email: reasoninginfo@reasoning.com