Executive Summary

Four years of analysis of the source code for the Linux operating system have revealed that the code has improved significantly. Coverity has combined two years of analysis work carried out in a commercial setting at Coverity, with four years of academic research conducted at the Computer Science Laboratory at Stanford University using an early prototype of Coverity's technology. The results show that the number of defects detected by the Coverity analysis system has decreased from over 2000 to less than 1000 between the years 2000 and 2004 while, during the same period of time, the source code has quadrupled in size and the power of Coverity's defect detection capabilities has increased markedly. We conclude using this data that the Linux kernel is a robust, secure system that has matured significantly.

About Coverity

Coverity was founded by Stanford professor Dr. Dawson Engler and four Ph.D. students in the Computer Science Laboratory at Stanford University. By combining a strong academic foundation with fast growing market traction experience, Coverity is able to leverage a combination of innovative ideas and enterprise requirements to build a source code analysis technology that is unparalleled in either research or industry.

Coverity Prevent™, our flagship product, is used by industry leaders such as Juniper Networks, Windriver Systems, Oracle, Synopsys, nVidia, and palmOne. Coverity currently has over 40 customers and has quadrupled in size over the last year. We continue to improve our analysis technology and open source software is a powerful validation for our new technology.

1 This product was formerly known as SWAT.
Coverity and Linux

Coverity's experience with the Linux source code began when the founding team was working at Stanford on a research project then known as the Meta Compiler Project. The project was focused on proving that compiler technologies could be successfully applied to find defects in robust software systems. The first published work on the Meta Compiler system was described to the research community in late 2000. In that publication, 511 bugs were reported, while the research for that publication uncovered over 2000 bugs in the Linux source code using the primitive analysis technology in our first experimental implementation of the analysis platform.

In 2001, a follow up presentation to the research community used the second iteration of the analysis platform to present a study of defect trends in the Linux operating system. In this study, over 1000 defects were detected in the Linux source code from version 2.4.1. At that time, the source code contained 1.6 million lines of code.

Another follow-on study illustrated a basic technique for detecting security flaws in Linux that uncovered 125 security flaws in the 2.4.12 Linux kernel, as well as substantial security flaws in an un-named commercial software system.

As Coverity's analysis technology matured, repeated analyses of the Linux kernel were performed as a means for validating the progress of the Coverity's technology. This work was punctuated by releases of security holes to the Linux kernel implementers in early 2004, and a July 2004 posting of the http://linuxbugs.coverity.com site that presented approximately 900 bugs in the Linux 2.6.4 kernel source code. At that time the Linux source code contained over 5 million lines of code. These bugs were released publicly to the Linux kernel implementers and many were fixed, although the exact number of subsequent fixes is still yet to be assessed. Several patches to the Linux kernel were applied in response to that release.

In sum, the number of defects uncovered in the Linux source code by Coverity's analysis decreased from year 2000 to 2004, a telling data point given that both the power (and range) of Coverity's defect detection capabilities and the actual size of the Linux source base increased significantly.

---


The Linux Study

We report here an updated analysis of the Linux kernel that uses the latest version of the Coverity analysis platform to analyze the Linux 2.6.9 kernel, which now contains over 5.5 million lines of code. These results represent an upgrade over those released in July, 2004 due to significant changes in the Coverity analysis system:

- Revamped interprocedural analysis that understands calling context and return context.
- Upgraded analyses that detect more stack and heap buffer overruns, memory leaks, and null pointer dereferences.
- Improved memory analysis capabilities to track complex flows of data throughout the source code.
- Improved value analysis capabilities to track buffer sizes and accesses more precisely.

Coverity's team regularly reports detected security flaws and other critical bugs to the Linux kernel implementers. In addition, we have made the detected bugs in the latest study available to the maintainers of the major components of the Linux kernel. In general, these bug reports are quickly converted into patches by the responsible implementers that remedy the defective source code. The patch is then released to the Linux kernel mailing list for vetting by a broader audience that helps ensure that each fix is, in fact, a complete fix, and that each fix does not introduce yet another problem. Patches are generally applied to the latest release candidate of the kernel by the maintainer for that candidate release within a matter of days of the release of the bug reports.
Results and Methodology

Coverity Prevent™ is a source code analysis system. Source code analysis is a unique approach to software quality for the following reasons:

- It is applied at compile time before a system ever runs.
- It does not require any test cases, manual testing, or user intervention.
- It pinpoints the location and root cause of each detected result.
- It can exercise all executable paths in a system, not just those covered by test cases.

Using this technology on the Linux 2.6.9 kernel, 950 defects were detected. The following table shows the raw counts broken down by bug type:

<table>
<thead>
<tr>
<th>Type</th>
<th>Total</th>
<th>Bug</th>
<th>Not Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td>103</td>
<td>33</td>
<td>4</td>
</tr>
<tr>
<td>Overrun Dynamic</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Overrun Static</td>
<td>119</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>Negative Returns</td>
<td>46</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Forward Null</td>
<td>327</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Deadcode</td>
<td>147</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Null Returns</td>
<td>70</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Resource Leak</td>
<td>112</td>
<td>9</td>
<td>20</td>
</tr>
<tr>
<td>Reverse Null</td>
<td>161</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Reverse Negative</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The bug types are described in the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td>Use of resources that are no longer available.</td>
<td>Data corruption, possible crash.</td>
</tr>
<tr>
<td>Overrun Dynamic</td>
<td>Buffer overrun in the heap.</td>
<td>Data corruption, possible crash, possible malicious attack.</td>
</tr>
<tr>
<td>Overrun Static</td>
<td>Buffer overrun in the stack.</td>
<td>Data corruption, possible crash, possible malicious attack.</td>
</tr>
<tr>
<td>Negative Returns</td>
<td>Buffer overrun using a negative offset in stack/heap.</td>
<td>Data corruption, possible crash, possible malicious attack.</td>
</tr>
<tr>
<td>Forward Null</td>
<td>Null pointer dereference.</td>
<td>System crash.</td>
</tr>
<tr>
<td>Deadcode</td>
<td>Unused code due to logic flaws.</td>
<td>Incorrect system functionality.</td>
</tr>
<tr>
<td>Null Returns</td>
<td>Null pointer dereference.</td>
<td>System crash.</td>
</tr>
<tr>
<td>Resource Leak</td>
<td>Leak of memory or other system resource.</td>
<td>Performance degradation, possible denial of service.</td>
</tr>
<tr>
<td>Reverse Null</td>
<td>Null pointer dereference or spurious Null check.</td>
<td>System crash or performance degradation.</td>
</tr>
<tr>
<td>Reverse Negative</td>
<td>Buffer overrun using a negative offset in stack/heap or spurious check against a negative value.</td>
<td>Data corruption, possible crash, possible malicious attack, or performance degradation.</td>
</tr>
</tbody>
</table>
The following graph shows the breakdown of bugs by location in the kernel source code:

Coverity's analysis system was applied to the Linux code base in three successive runs. The final run used a combination of hand configuration for Linux-specific interfaces and statistical analysis of Linux interfaces to obtain the best result set.

The results were inspected at random by Coverity's engineers to determine which reports are actual defects in the source code that should be fixed and which results should be ignored. Those that should be ignored are often invalid due to one of the following factors:

- Incomplete configuration for complex interfaces.
- Idioms in the Linux source code that are not analyzed, including complex data structures, inline assembler, etc..
- Complexities in the source code that the analysis tool cannot model accurately.

A more complete validation of the results will be achieved by reporting the bugs directly to the responsible Linux implementers and tracking which bugs are fixed and which are ignored by those responsible for the source code.

The reported bug count of 950 was computed by using the observed rate of false error reports to estimate what percentage of the detected reports might be invalid. In cases where zero results were inspected, a historical false positive rate of 20% was used to estimate the number of valid results.

Coverity Prevent™ can look through all paths in the code, cross procedure and file boundaries, and avoid any assumptions beyond the semantics of the programming language. Prevent thus has the broad vision to understand what the corner cases are and how one might exploit them.
Coverity has developed the first system that can identify defects with a combination of precision, speed, and accuracy across millions of lines of code that satisfies the most demanding industry customers as well as the fast paced open source community.

Source Code Analysis and Security

Source code analysis can effectively address classes of errors that testing cannot, particularly with respect to security holes. Corner cases and unintended inputs are the cause of most security vulnerabilities that are introduced by coding mistakes. Coverity Prevent™ can look through all paths in the code, cross procedure and file boundaries, and avoid any assumptions beyond the semantics of the programming language. Prevent thus has the broad vision to understand what the corner cases are and how one might exploit them.

In particular, Coverity Prevent™ currently detects several types of buffer overruns in the heap and the stack. Those results marked as "Negative Returns" and "Reverse Negative" may have the most devastating security implications because they are often triggered by rare failure cases that almost never happen in normal system operation. The goal of a malicious attacker, though, is to drive the system into an unexpected state, which can, in some cases, trigger these disastrous buffer overruns.

In general, the potential of source code analysis in the security arena is undeniable. Early tests at Coverity have shown that Coverity's analysis technology can be used to find a variety of security holes that are triggered by malicious user input to a system. Initial tests on the Linux kernel and the OpenBSD kernel have revealed several security flaws. (REF Links) Development on these analyses is ongoing at Coverity.

The Evolution of Source Code Analysis Technology

Compile time source code analysis has long remained more of a promise than a reality. Coverity has developed the first system that can identify defects with a combination of precision, speed, and accuracy across millions of lines of code that satisfies the most demanding industry customers as well as the fast paced open source community. Early attempts at static source code analysis focused on software verification. While these efforts yielded tools, the domains of those tools were inevitably limited to either restrictive, custom programming languages, or very small programs that do not reflect the complexity of today's commercial software.

Coverity Prevent™ represents a breakthrough in the source code analysis community by applying sophisticated compiler optimization techniques combined with an innovative system architecture to produce the most effective analysis system ever built. Coverity Prevent™ is designed as a generic analysis platform with modular plug-ins that carry out specific simulations of the source code and report specific types of defects. Because of this modular design, each analysis is finely tuned to the type of bug that it detects, allowing Coverity's system to achieve a false positive rate that is the overall lowest amongst source code analysis systems.

The Impact of Source Code Analysis

In today's software industry, the costs of software defects are disastrous. A May 2002 report prepared for the National Institute of Standards and Technologies (NIST) estimates the annual cost of software defects in the
United States as $59.5 billion. Defects that go undetected until after a product has shipped can cost the producers of that software many tens of thousands of dollars each to address and patch. The cost suffered by the users of that product is often orders of magnitude higher. For every thousand lines of code developed by commercial software makers or corporate programmers there could be as many as 20 to 30 bugs.\(^5\)

The root cause of these astronomical costs is the increasing complexity demands for today's software. Modern server side applications such as operating systems, application servers, and databases usually contain hundreds of thousands if not millions of lines of source code.

The implications of shortcomings in the software quality process mean that most developers spend approximately 80% of their time fixing existing code as opposed to producing new code. Also, about 40-50% of the effort on current software projects is spent on avoidable rework, effort spent fixing difficulties with the software that could have been avoided or discovered earlier and less expensively.\(^6\) The drag in time to market and the costs of disastrous software releases become imposing burdens on almost all software development organizations. The fundamental need for a better software quality process is immediately apparent to any organization that strives to build robust, secure products.

Source code analysis is beginning to take hold in the market as an alternative to traditional testing techniques. Defects detected with source code analysis are detected earlier in the development cycle are significantly easier to fix and are often outside the boundaries of what traditional testing can detect. Most importantly, the cost of fixing defects during development is significantly cheaper than addressing them once software is deployed in the field. Early adopters of this technology are already realizing the benefits of delivering more stable, secure products with an accelerated development schedule.

**Coverity Code Audits**

Coverity offers free code audits analogous to what is contained in this report. If you are interested, please email sales@coverity.com or please visit www.coverity.com.

---

\(^5\) [http://www.eweek.com/article2/0,1759,1544234,00.asp](http://www.eweek.com/article2/0,1759,1544234,00.asp)