Source Code and Binary Analysis of Software Defects

Daniel Quinlan
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA
dquinlan@llnl.gov

Thomas Panas
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA
panas@llnl.gov

ABSTRACT
This extended abstract presents the techniques to identify a selected set of software defects (bugs, bad practices, etc.) within both source code and binary executables. We present the results from six different static analysis tests applied on both the source code and the binary executables (with and without optimization) on three different applications. We compare the precision of the static analysis results from the source code and the binary executable forms of the same software. Ideally the results from an analysis of source code and its binary would be identical, but in practice the source code and binary representation cause slightly different techniques to be used with different amounts and types of information readily available.

Our work defines a few defect analyses to support what might later be a larger collection of analyses. Our goal is to more thoroughly evaluate software quality and eliminate, as much as possible, the classic asymmetry of information about software, specifically quality as understood by the software developer vs. the software user. It is not well studied how static analysis of source code and binaries are related for purposes of evaluating general quality and our work is focused in this direction; much less are the tools for such work openly available. Our work also presents an open framework well suited for identifying general software properties of both source code and binary executables.

1. INTRODUCTION
Within this extended abstract we define software defects as faults that are introduced unintentionally into computer programs, preventing those programs from behaving correctly. They commonly arise through bad design or bad implementation practices. Defects lead either to the malfunction of a software system (software bug)—meaning the program behaves unpredictably at run-time—or to security flaws which might later be exploited (software vulnerability). Unfortunately, software defects are omnipresent in almost all software developed today, and even worse, they can lead to the unavailability of the most critical systems such as emergency, financial, electric power or military systems. For instance, a Los Angeles Times article reported that the Red Team hackers hit the jackpot when they broke into networks that direct the 911 emergency response systems. Above all, software defects are expensive. According to a 2002 National Institute of Standards and Technology study, software errors cost the U.S. economy an estimated 59.5 USD billion annually.

For these reasons it is crucial to evaluate the quality of software generally and detect and prevent software defects early. Many tools for software defect detection exist but almost all try to find defects on source code. However, often the source code is unavailable. It is nevertheless critical to allow software users to have insight into the quality of software that they purchase. Detecting defects in binaries is more difficult, but worth pursuing because, if possible, it could lead to a fundamental shift in the transparency of quality in software. The analysis of software can be done both dynamically or statically; our work focuses on static analysis. The majority of commercial tools use static analyses to evaluate the quality of source code. The use of static analysis techniques on binary executables is generally referred to as binary analysis. Where binary analysis can be used to detect defects, it permits an evaluation of software quality. Binary analysis may be useful in grading aspects of (competing) software products where source code can not (easily) be inspected.

A long-term goal is to define a new level of transparency in the evaluation of software and specifically remove the asymmetry of information about software quality. The asymmetry is in the difference between what the software developers know, or think they know, about the software quality and what the user can discover. Through greater transparency, software with better quality can ultimately be rewarded (e.g. through increased prices to reflect the value of added quality). This situation in economics is known commonly as the lemon law.

We developed two tools, Compass and BinQ based on the ROSE compiler infrastructure that analyze source code and binaries, respectively. Both tools and ROSE are available at www.roseCompiler.org and are part of a larger project at LLNL to support the general analysis and transformation of software. Within each tool, we implemented program analyses that search for security flaws and bad coding styles. We implemented conceptionally equivalent defect detectors with implementations (for source code and for binary analysis) that are as similar as possible. This is feasible mainly because all analyses (for source code and binary) use the same same infrastructure (ROSE). Using ROSE we have a significant opportunity to unify how the general analysis of software (both source code and binaries) can be explored.

In Section 2 we introduce our software analysis tools: ROSE, Compass and BinQ. Our results are briefly reported in Section 3, with some discussion in Section 4, the summary is appropriate for an extended abstract obviously many details are left out. Section 5 presents some conclusions.
2. TOOLS

In this section we describe the ROSE compiler infrastructure, as well as Compass and BinQ—tools that were built using ROSE. Compass was built initially for source code analysis and has been extended to support source code and binary analysis uniformly. BinQ is specific for binary analysis, and many of the forms of analysis are moved to Compass once they become more mature. Both tools were used to generate the results shown in Section 3.

2.1 ROSE

ROSE is a compiler infrastructure that is open source, BSD licensed and freely available and developed principally at LLNL. ROSE is unique because it is a source-to-source framework. It supports research work on compiler transformations, especially optimizations, and general analysis of source code and binary executables. As a research tool infrastructure, it allows any developers or scientists to utilize prepackaged compiler expertise as libraries to construct custom analysis and transformation tools. Literally anyone with programming skills in C++ (ROSE is written in C++) can build their own customized software analysis and optimization tools for applications in C, C++, PHP, or Fortran 2003, OpenMP, and UPC. ROSE was developed for non-experts and with the primary goal to be easy to use. In general, specialized compiler expertise expands the range of what can be done using ROSE.

Figure 1 illustrates the general approach of ROSE. ROSE provides interfaces for the user to perform compiler specific tasks, such as the reading-in (parsing) of C/C++ or Fortran source code and the construction of an internal intermediate representation (IR). The parsing itself is performed by utilizing well-established frontends, such as the Edison Design Group (EDG) C++ front-end for C and C++ and the Open Fortran Parser (OFP) for Fortran. To support optimization of scientific codes in DOE, ROSE handles C (C89,C99), C++, Fortran 2003 (including P66, F77, F90/95), OpenMP, and UPC. ROSE was developed for non-experts and with the primary goal to be easy to use. In general, specialized compiler expertise expands the range of what can be done using ROSE.

void visit(SgNode* node) {
  SgCastExp* cast = isSgCastExp(node);
  if (cast && cast->get_cast_type() == SgCastExp::e_C_style_cast)
    output->addOutput(new DetectorOutput(node));
}

Figure 2: Conceptual Example of ROSE AST.

2.2 Compass

Compass is a tool for program quality and security analysis. Because it is based on ROSE, it allows users to implement their own program checkers to locate and report software defects. Documentation of some of the 100 checkers that we have implemented can be found on the ROSE Webpage. Our checkers are based on common programming faults or known security risks reported e.g. within the Common Weakness Enumeration (CWE) or by the Computer Emergency Response Team (CERT). The focus of Compass is not to define better and faster analyses for software defects but rather to allow users who do not necessarily have compiler backgrounds to develop their own expert defect detectors.

Detector rules for source code analysis can simply be specified as programs operating on the ROSE AST. Each rule is specified separately, and thus the rules can be evaluated independently. More complex rules may be defined on either the control flow graph, system dependence graph, call graph, class hierarchy graph, data flow graph or combinations of those.

As an example, consider the following implementation of a simple Compass detector:

As an example, consider the following implementation of a simple Compass detector:
The above code shows our implementation of the CERT Secure Coding Standard specification referred to as EXP00-A. Do not use C-style casts. Although C++ allows C-style casts, from a security perspective it is safer to use C++-style casts that allow for more compiler checking. This defect has CERT severity level 3 (high) because unchecked type errors could allow attackers to exploit the code. Our implementation of this defect resides in the visit function above, representing a visitor pattern: this method is called on each node of the program’s AST. If the node is a cast and it is of type e_C_style_cast, a defect is detected and reported. Because ROSE has type information available in the AST, both implicit and explicit casts are detected. Full type information is available even for sophisticated uses of C++ templates. In recent work, we have also evaluated aspects of parallel distributed memory AST traversals including defect detection using up to 300 processors.

Compass currently supports five types of defect reporting. Besides textual output, we have connected Compass to Emacs, Vim, Eclipse, and the QRose GUI (QRose is a Qt-based GUI library for ROSE developed in collaboration with Gabriel Coutinho of Imperial College London). In particular, Emacs’s Flymake can run Compass in the background while a user is working, allowing a real-time display of Compass analysis results. For Vim, a compiler plugin has been provided for users to work with Vim’s QuickFix commands to highlight source code lines containing rule violations. Our Eclipse plug-in for Compass is based on the C++ ROSE implementation, connected to Java using SWIG. Figure 3 shows a screen shot of software defect analysis using Compass with Eclipse. Defect analysis results are represented in the problem tab in the lower corner of the figure.

![Figure 3: Compass as an Eclipse Plug-in.](image)

2.3 BinQ

Recently, ROSE has been extended with the capability to handle software binaries, cf. Figure 1. For this, users may utilize one of two frontends: IDA-Pro the industry standard for interactive disassembly or the in-house developed disassembler that is now part of ROSE. IDAPro supports many different processors under Linux and Windows while our own disassembler currently only supports the x86 and ARM architectures. The disassembled binaries are represented in the ROSE IR in the same way as source code when parsed. This has the advantage that mechanisms for source code analysis can be re-used for binaries, such as AST traversals, integrity checks, documentation generation, IR node generation, etc. The current implementation supports binary analysis, transformation and rewriting.

BinQ is our experimental tool for software binaries using ROSE. We use BinQ, cf. Figure 4 to develop binary analyses and to improve our binary infrastructure in ROSE. Once individual binary analyses mature, they will be ported to Compass—allowing Compass to analyze source code and binaries.

For the rest of the extended abstract we assume that the binaries we analyze are not obfuscated or packed. In fact, since we have the source code of our “test cases” (objects) available, we compile the sources themselves into binaries using in addition symbol information. Also, we use our own recursive disassembler in ROSE instead of IDA-Pro.

3. RESULTS

Table 1 shows our results for applying source code and binary analyses on mailx, SDCC and RTED. The results are presented, but not explained in detail since this is an extended abstract. We focus on source code and (unoptimized) binary results and include a brief discussion in the next section.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Program</th>
<th>Source</th>
<th>Binary</th>
<th>Binary -O2</th>
</tr>
</thead>
<tbody>
<tr>
<td># Unsafe Functions</td>
<td>mailx</td>
<td>194</td>
<td>211</td>
<td>157</td>
</tr>
<tr>
<td></td>
<td>SDCC</td>
<td>370</td>
<td>385</td>
<td>274</td>
</tr>
<tr>
<td></td>
<td>RTED</td>
<td>49</td>
<td>49</td>
<td>19</td>
</tr>
<tr>
<td>#Malloc without Free</td>
<td>mailx</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>SDCC</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RTED</td>
<td>45</td>
<td>15</td>
<td>34</td>
</tr>
<tr>
<td>#Dangling Pointers</td>
<td>mailx</td>
<td>183</td>
<td>160</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>SDCC</td>
<td>18</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>RTED</td>
<td>275</td>
<td>247</td>
<td>122</td>
</tr>
<tr>
<td>#Uninitialized Pointers</td>
<td>mailx</td>
<td>1,873</td>
<td>222</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>SDCC</td>
<td>4,503</td>
<td>106</td>
<td>1,328</td>
</tr>
<tr>
<td></td>
<td>RTED</td>
<td>2,464</td>
<td>61</td>
<td>139</td>
</tr>
<tr>
<td>#Buffer Overflows</td>
<td>mailx</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>SDCC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>RTED</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#Complex Functions</td>
<td>mailx</td>
<td>45</td>
<td>40</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>SDCC</td>
<td>265</td>
<td>254</td>
<td>288</td>
</tr>
<tr>
<td></td>
<td>RTED</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sum (total)</td>
<td>mailx</td>
<td>2,277</td>
<td>635</td>
<td>422</td>
</tr>
<tr>
<td></td>
<td>SDCC</td>
<td>5,158</td>
<td>766</td>
<td>1,898</td>
</tr>
<tr>
<td></td>
<td>RTED</td>
<td>2,834</td>
<td>372</td>
<td>314</td>
</tr>
<tr>
<td>Sum (without uninit. pointers)</td>
<td>mailx</td>
<td>404</td>
<td>413</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>SDCC</td>
<td>656</td>
<td>660</td>
<td>570</td>
</tr>
<tr>
<td></td>
<td>RTED</td>
<td>370</td>
<td>311</td>
<td>175</td>
</tr>
</tbody>
</table>

Table 1: Source Code and Binary Analysis Results.

4. DISCUSSION

Our experiments reveal interesting insights about the precision of our analyses, why results between source and binary differ and why some defects can not be detected (in either the source or binary). The last row of Table 1 reports that for the sum of the defects (excluding the least representative case for Uninitialized Pointers), the differences between source code and binary analysis results for each of the three applications are approximately %1, %2, and %20; respectively. An investigation of the differences also suggests improvements to our analyses so that defects can be found even more precisely.

In particular, we have detected the following issues:

**# Unsafe Functions.**

Our binary frontend needs to be improved to detect function boundaries better, this is being done currently. This will improve the precision of our binary analysis. Optimizations automatically performed by the compiler, such as conversion from sprintf to memcpy and atoI to strtol are not something we can work on to improve our analyses. This is therefore a fundamental difference that will always lead to slightly different results between
source code and binary defects. It may be possible, for a specific compiler, to predict where in the source code such transformations are likely.

# Malloc without Free.

We believe that if we invest more effort into our analysis implementations, especially the source code analysis for the malloc without free analysis, we can detect the exact same number of defects for both, source code and binary.

# Dangling Pointers.

We have detected problems within both our source code analysis and binary analysis. But within both cases, we can improve the precision if we elaborate on our analyses. For the source code, we need to be able to track pointers better and for the binary, we will have to evaluate expressions.

# Uninitialized Pointers.

We can conclude that it is possible to detect uninitialized pointers in source code easily when we want the pointers to be initialized at their declaration. Since declarations don’t exist in the binary, a similar approach in the binary does not make sense. But the source code analysis could and should be more similar to that for the binary (detecting uses before defs on the control flow).

# Buffer Overflows.

Buffer overflow detection may require additional techniques to deliver precise results. This works well for source code and binary analysis for our example tools. However, the analyses would quickly be challenged if we had to evaluate expressions, follow pointers or in the worst case handle user input—for which we might not get any results at all.

# Complex Functions.

There is a correlation between source code and binary defects regarding the cyclomatic complexity. This could indicate that complexity analyses on binaries are possible even that distinctions between different kinds of conditions can not be made. Additional investigations will be required on different types of code to evaluate if this metric computed on the binary can be correlated to the source code. We know that sophisticated loop optimizations in scientific code are likely to generate extremely different results for this metric. But it might be reasonable on unoptimized code. In our results, we have noticed that the optimized and unoptimized cases are similar and that they correlate well with the source code results.

5. CONCLUSION AND FUTURE WORK

We have used a collection of static forms of analysis written using the same infrastructure to analyze both source code and the binaries built from the source code (with and without optimization). From this we have not always obtained identical results, but we have obtained similar results. Where they have not been the same this has lead to interesting examples of how binary executables are fundamentally different from source code. In a few cases this has helped us identify bugs in our own analyses. Where they are the same it is often because the binary is not so different from the source code for that specific analysis. Based on our experience so far, we expect a larger collection of defect analyses could be defined and they could be implemented to correlate quality in the source code to that measured indirectly using only the binary executable.

Currently, if it is possible to measure the quality of software it is most likely accessible via only source code analysis. This is not a problem for open source code, but a majority of software is closed source. Because only software developers have the source code, the results are held privately, if they are even known. This denies the user any mechanism to obtain an independent analysis of software quality and forms the fundamental asymmetry that limits the ability of software quality to be priced (and rewarded). We expect that the simple existence of mechanisms to independently measure the quality of software (transparency) may have far reaching effects to improve the quality of software for everyone. We have selected a collection of static analyses and demonstrated that they can be readily implemented for both source code and binary executables and that the results. While not identical, the results are similar enough to support a statistical evaluation of the source code indirectly from only the binary. We have not tried to assess overall software quality since that would presumably require many more kinds of static analyses than the few we have implemented (and a theory for now the results might be assembled to a measure of quality).