Techniques for Software Quality Analysis of Binaries: Applied to Windows and Linux

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

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

ABSTRACT

In this paper we present our efforts to measure different quality aspects of large-scale, binary software. We apply four well established metrics to binary versions of Windows and Debian Linux, analyze our results and discuss our observations. It is surprising to see that our metrics, which search for well known bad coding habits, result with so many violations. It appears that although bad and insecure software development practices are well understood and documented, in practice, recommended coding styles are not followed. Our work evaluates binary versions of software, allowing us to inspect software quality without the need of source code. We believe that this approach, if successful, could lead in the future to better priced software. This is because the quality of software bought today is not transparent to its users.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms

Security, Measurement

Keywords

Static Analysis, Binary Analysis, Software Quality

1. INTRODUCTION

Bad software coding practices often lead to vulnerable software, which then is either exploited by adversaries or more prone to fail at runtime. For this reason, bad coding habits (styles) are documented in the literature, c.f. [2, 6]. These documents elaborate not only on bad practices and possible vulnerabilities, but also on the severity of using such practices. Preferably, bad coding styles should be avoided so that software with high quality can be developed and deployed. The evaluation of software quality is an attempt to characterize and measure the prevalence of such flaws. Because software developers are used to looking at code, the source code is frequently considered the most relevant medium in which to measure software quality. Far more time and significantly greater expertise is required to identify the same bad coding styles in software binaries. The short availability of binary expertise and the difficultly associated with it have led many to believe that the measurement of software quality directly in the binary is intractable. Our work shows that automated tests can be developed that measure the same software flaws in either the source code or the binary using substantially similar techniques. Our work uses the ROSE [10] framework which substantially unifies the analysis of both source code and binaries providing an approach to test this idea.

A specific goal is to remove the asymmetry of information associated with the measurement of quality of source code owned by the developer and COTS software more commonly available to users. Properly measured this would reward developers and permit price to match quality; resulting in better software.

We developed *BinQ*, which is based on the *ROSE* compiler infrastructure [10], to analyze software binaries. Within BinQ we implemented literature based program analyses that search for possible security flaws and bad coding styles. In Section 2 we introduce our software analysis tools: ROSE and BinQ. We selected and implemented four analysis metrics, c.f. Section 3. Analysis results are reported in Section 4. Verification is discussed in Section 5. Related work and conclusions are found in Section 6 and Section 7, respectively.

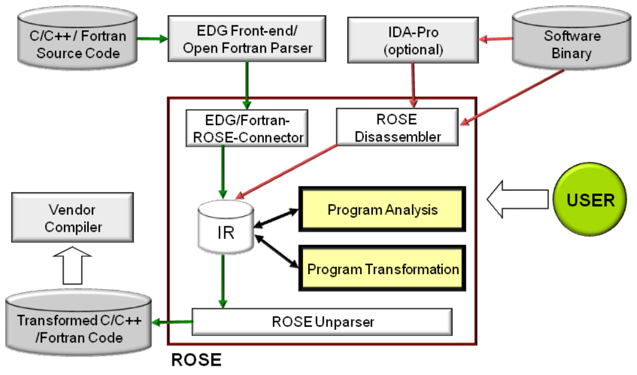


Figure 1: ROSE, a customizable compiler infrastructure

2. TOOLS

ROSE is a source-to-source compiler infrastructure that is open source, BSD licensed and freely available. ROSE supports research work on compiler transformations, especially optimizations, and general analysis of source code and binary executables. ROSE was developed for experts and non-experts to build their own customized software analysis and optimization tools.

Figure 1 illustrates the general approach of ROSE. ROSE provides interfaces for the user to perform compiler specific tasks, such as the 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 [4] for C and C++ and the Open Fortran Parser (OFP) [8] for Fortran. To support optimization of scientific codes in DOE, ROSE handles C (C89,C99), C++, Fortran 2003 (including F66, F77, F90/95), OpenMP, and UPC. To support general binary analysis ROSE supports the x86, PowerPC, and ARM instruction sets using a number of specific binary file formats: ELF, PE, LE, NE, and MS-DOS.

All information about the source code and the binary is represented in an easily traversed *abstract syntax tree* (AST). The generated ROSE IR includes an AST, symbol tables holding types, comments, pre-processor information, etc. The IR is rich enough so that the original program can be faithfully represented and reproduced. Once user defined analyses and transformations have been implemented and applied, the ROSE Unparser, cf. Figure 1, outputs the new (optimized) program back to source code in the original source language with all original source level details (including comments and C preprocessor control structures) preserved. Optionally, vendor compilers can be used to compile the transformed source code into executables for different platforms.

For binary handling, ROSE may be utilized using one of two frontends, cf. Figure 1: IDA-Pro [3] the industry standard for interactive disassembly or the in-house developed recursive disassembler (part of ROSE). IDA-Pro supports many different processors under Linux and Windows, our own disassembler currently supports the x86, Power-PC, and ARM architectures. A distinguishing point is that our parsing of the binary includes the whole binary file format (all sections, including dwarf) and all details required to reconstruct the original binary executable; making the full representation available for analysis. 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.

3. ANALYSES

In order to evaluate bad coding styles on binaries using our binary analysis capability within ROSE, we chose four well established metrics from the literature [2, 5, 6] that detect bad coding styles in source code. We implemented these four well defined coding violation specifications as "checkers" (analyses) to detect bad coding styles in binaries. Initially, we chose simpler metrics that do not require data-flow analysis, in order to properly evaluate our ongoing work with binaries. Again, for this experiment, we merely re-implemented well defined bad coding styles. It is not up to us to judge about the appropriateness of the metrics. Our goal is to establish a tool that allows us to write arbitrary analyses on binaries - including data-flow analyses in the future. Our selection and implementation of analyses is described next.

3.1 # Unsafe Functions

Definition: Our first analysis detects calls to possibly unsafe functions. We define unsafe functions as functions with possible side effects. According to CERT [2] such functions should be replaced by their "safe" counterparts.

Implementation: We define, in accordance to literature [2], the following functions and hence calls to them as unsafe:

- vfork: *Do not use this function in POSIX program.*
- sprintf, scanf, sscanf, gets, strcpy, _mbscpy, lstrcat, memcpy, strcat: *May cause buffer overflow.*
- rand: *Does not produce high-quality random numbers.*
- rewind: *Implement fseek over rewind.*
- atoi, atol, atoll, atof: *Use strtol, strtoll, strtod for converting strings to numbers.*

Using ROSE, the implementation of the analyses is merely a traversal of the programs' AST. Whenever a AST node is traversed that represents a call to a function (for both source code and binary), the name of the function is resolved and checked against the list above. ROSE offers the ability to resolve function names by using a files input table. However, in our experiment all symbols are provided by IDA-Pro.

3.2 # Complex Functions

Definition: *Cyclomatic Complexity* computes the number of control branches within a function [5]. Branches are defined through if-conditions, switch statements, goto statements, while loops, etc.

Implementation: This analysis counts the number of unconditional and conditional control transfer instructions within each function. This includes instructions such as *x86_jmp*, *x86_loop*, *x86_ja*, *x86_je*, etc. The threshold that we assigned for this analysis is 45, i.e. if the number of conditions is 45 or more, we flag the function to be complex.

Our threshold is arbitrarily chosen. In practice 20 is a common value used in industry for source code (McCabe's Complexity Metric) whereas 45 should merely reflect the fact that there are more control transfer instructions within a binary than within source code. Within our experiment we have found that 45 is a good binary correlation value to 20 used in source code. However, this value can be individually adjusted just as it is adjusted for source code purposes.

3.3 # Malloc without Free

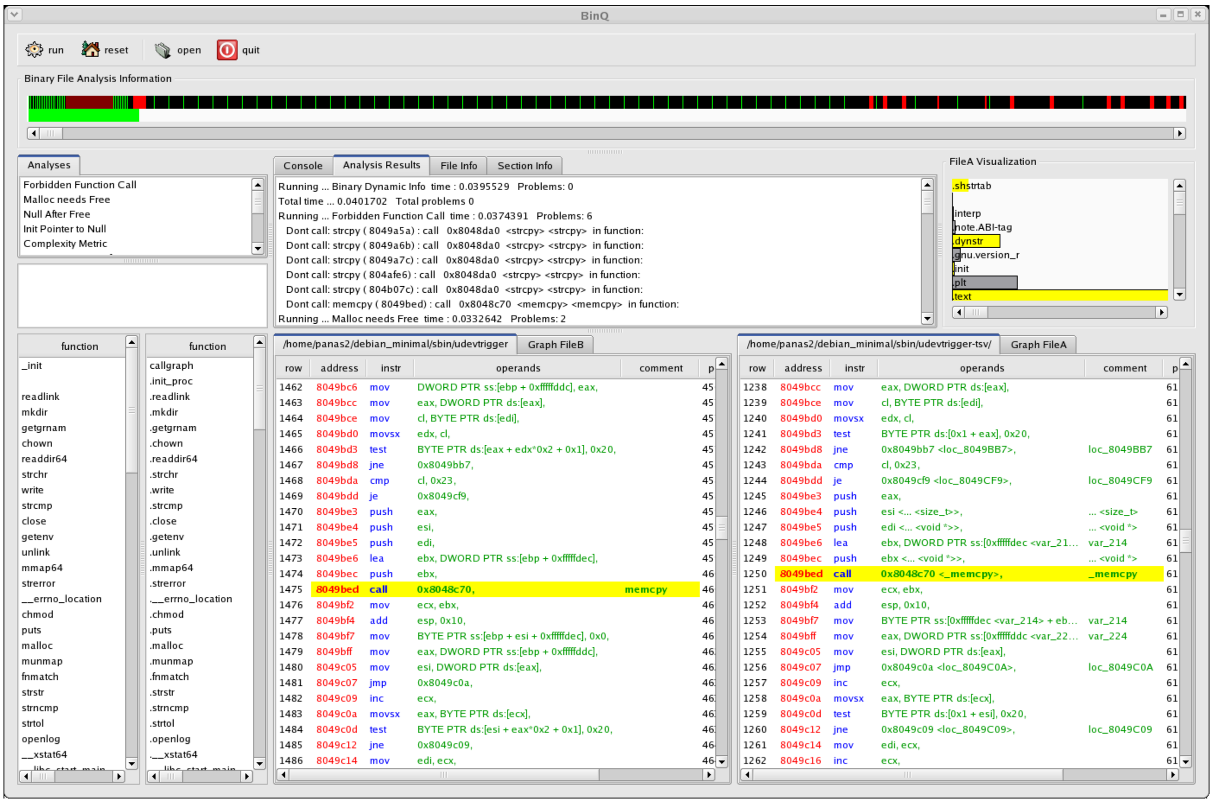
Definition: According to CERT [2] it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.

Implementation: Initially, our algorithm traverses the AST and looks for AST nodes representing a `call` instruction. When a call instruction is found, its symbols are resolved and hereby also the name of the function that is called. If the function to be called is `malloc` or `calloc`, then we have found a candidate. As an example look at the following source code and its assembly form.

```
1 int main(int argc, char* argv) {
2  int* arr = malloc( sizeof(int)*10);
3  free(arr);
4 }
```

```
0x4004f8: push   rbp
0x4004f9: mov    rbp, rsp
0x4004fc: sub    rsp, 0x20
0x400500: mov    DWORD PTR ds:[rbp + 0xffffffec], edi
0x400503: mov    QWORD PTR ds:[rbp + 0xffffffe0], rsi
0x400507: mov    edi, 0x28
0x40050c: call   0x400420       %% call to malloc
0x400511: mov    QWORD PTR ds:[rbp + 0xfffffff8], rax
0x400515: mov    rdi, QWORD PTR ds:[rbp + 0xfffffff8]
0x400519: call   0x400430       %% call to free
```

In the assembly representation, the function header for main is represented at addresses `0x4004f8`, `0x4004f9` and `0x4004fc`. At address `0x40050c` is the call to `malloc`. The size of the allocation is encoded at address `0x400507`, namely `0x28` heximal (hex) or 40 decimal (dec). Thereafter, at address `0x400511` the result from register `rax` is stored in memory at location `ds:[rbp + 0xfffffff8]`. This is the location of our pointer variable `arr` in line 2. Finally, `free` is called at address `0x400519` where the argument is stored in rdi, namely memory address `ds:[rbp +`



```
Running ... Binary Dynamic Info  time : 0.0395529   Problems: 0
Total time ... 0.0401702   Total problems 0
Running ... Forbidden Function Call  time : 0.0374391   Problems: 6
  Dont call: strcpy ( 8049a5a) : call   0x8048da0  <strcpy> <strcpy>  in function:
  Dont call: strcpy ( 8049a6b) : call   0x8048da0  <strcpy> <strcpy>  in function:
  Dont call: strcpy ( 8049a7c) : call   0x8048da0  <strcpy> <strcpy>  in function:
  Dont call: strcpy ( 804afe6) : call   0x8048da0  <strcpy> <strcpy>  in function:
  Dont call: strcpy ( 804b07c) : call   0x8048da0  <strcpy> <strcpy>  in function:
  Dont call: memcpy ( 8049bed) : call   0x8048c70  <memcpy> <memcpy>  in function:
Running ... Malloc needs Free  time : 0.0332642   Problems: 2
```

```
function               function
_init                  callgraph
                       .init_proc
readlink               .readlink
mkdir                  .mkdir
getgrnam               .getgrnam
chown                  .chown
readdir64              .readdir64
strchr                 .strchr
write                  .write
strcmp                 .strcmp
close                  .close
getenv                 .getenv
unlink                 .unlink
mmap64                 .mmap64
strerror               .strerror
__errno_location       .__errno_location
chmod                  .chmod
puts                   .puts
malloc                 .malloc
munmap                 .munmap
fnmatch                .fnmatch
strstr                 .strstr
strncmp                .strncmp
strtol                 .strtol
openlog                .openlog
__xstat64              .__xstat64
```

```
row   address   instr   operands                                        comment   p
1462  8049bc6   mov     DWORD PTR ss:[ebp + 0xfffffddc], eax,                     45
1463  8049bcc   mov     eax, DWORD PTR ds:[eax],                                  45
1464  8049bce   mov     cl, BYTE PTR ds:[edi],                                    45
1465  8049bd0   movsx   edx, cl,                                                  45
1466  8049bd3   test    BYTE PTR ds:[eax + edx*0x2 + 0x1], 0x20,                  45
1467  8049bd8   jne     0x8049bb7,                                                45
1468  8049bda   cmp     cl, 0x23,                                                 45
1469  8049bdd   je      0x8049cf9,                                                45
1470  8049be3   push    eax,                                                      45
1471  8049be4   push    esi,                                                      45
1472  8049be5   push    edi,                                                      45
1473  8049be6   lea     ebx, DWORD PTR ss:[ebp + 0xfffffdec],                     45
1474  8049bec   push    ebx,                                                      46
1475  8049bed   call    0x8048c70,                                      memcpy    46
1476  8049bf2   mov     ecx, ebx,                                                 46
1477  8049bf4   add     esp, 0x10,                                                46
1478  8049bf7   mov     BYTE PTR ss:[ebp + esi + 0xfffffdec], 0x0,                46
1479  8049bff   mov     eax, DWORD PTR ss:[ebp + 0xfffffddc],                     46
1480  8049c05   mov     esi, DWORD PTR ds:[eax],                                  46
1481  8049c07   jmp     0x8049c0a,                                                46
1482  8049c09   inc     ecx,                                                      46
1483  8049c0a   movsx   eax, BYTE PTR ds:[ecx],                                   46
1484  8049c0d   test    BYTE PTR ds:[esi + eax*0x2 + 0x1], 0x20,                  46
1485  8049c12   jne     0x8049c09,                                                46
1486  8049c14   mov     edi, ecx,                                                 46
```

```
row   address   instr   operands                                        comment        p
1238  8049bcc   mov     eax, DWORD PTR ds:[eax],                                       61
1239  8049bce   mov     cl, BYTE PTR ds:[edi],                                         61
1240  8049bd0   movsx   edx, cl,                                                       61
1241  8049bd3   test    BYTE PTR ds:[0x1 + eax], 0x20,                                 61
1242  8049bd8   jne     0x8049bb7 <loc_8049BB7>,                        loc_8049BB7    61
1243  8049bda   cmp     cl, 0x23,                                                      61
1244  8049bdd   je      0x8049cf9 <loc_8049CF9>,                        loc_8049CF9    61
1245  8049be3   push    eax,                                                           61
1246  8049be4   push    esi <... <size_t>>,                             ... <size_t>   61
1247  8049be5   push    edi <... <void *>>,                             ... <void *>   61
1248  8049be6   lea     ebx, DWORD PTR ss:[0xfffffdec <var_21...        var_214        61
1249  8049bec   push    ebx <... <void *>>,                             ... <void *>   61
1250  8049bed   call    0x8048c70 <_memcpy>,                            _memcpy        61
1251  8049bf2   mov     ecx, ebx,                                                      61
1252  8049bf4   add     esp, 0x10,                                                     61
1253  8049bf7   mov     BYTE PTR ss:[0xfffffdec <var_214> + eb...       var_214        61
1254  8049bff   mov     eax, DWORD PTR ss:[0xfffffddc <var_22...        var_224        61
1255  8049c05   mov     esi, DWORD PTR ds:[eax],                                       61
1256  8049c07   jmp     0x8049c0a <loc_8049C0A>,                        loc_8049C0A    61
1257  8049c09   inc     ecx,                                                           61
1258  8049c0a   movsx   eax, BYTE PTR ds:[ecx],                                        61
1259  8049c0d   test    BYTE PTR ds:[0x1 + esi], 0x20,                                 61
1260  8049c12   jne     0x8049c09 <loc_8049C09>,                        loc_8049C09    61
1261  8049c14   mov     edi, ecx,                                                      61
1262  8049c16   inc     ecx,                                                           61
```

Figure 2: BinQ - Our binary analysis tool showing an unsafe function call to memcpy in /sbin/udevtrigger (Linux). Lower left using our disassembly and lower right disassembly from IDA-Pro.

`0xfffffff8]`. This address represents our variable `arr` and so we know that `free` was called with `arr` as an argument.

Therefore, to come back to our algorithm, once a `malloc` call is found, we need to go forward in the programs control flow and detect a `mov` instruction that moves the register `rax` (representing the result of malloc) into memory. We remember the memory location and traverse the function further according to its control flow. If we find a call to `free`, we need to traverse the control flow backwards to determine the last movement of memory to a register (parameter to free). Thereafter, we compare the memory location against the one we remembered and if they match, we have found a matching `malloc` and `free`. Otherwise, we report a coding style violation.

3.4 # Dangling Pointers (after Free)

Definition: Pointers that are not set to NULL after the function `free` is called. Dangling pointers can lead to exploitable double-free and access-freed-memory vulnerabilities [2]. A simple yet effective way to eliminate dangling pointers and avoid many memory related vulnerabilities is to set pointers to NULL after they have been freed.

Implementation: The binary analysis traverses the ROSE AST and checks every `call` instruction for a call to `free`. If found, we follow the control flow backwards (from that node) to find the memory location that corresponds to the variable that is being freed. Thereafter, we follow the control flow forward (from the same `call` node) to find `mov` operations. We are looking for a `mov mem,val` instruction that copies a value, in our case NULL (or 0) to the memory location of our predetermined variable (argument to `free`). If no corresponding `mov mem,0` is found, then we have detected a violation of this metric.

4. RESULTS & DISCUSSION

We applied BinQ on Windows XP and Debian Minimal (Linux). In both cases we used IDA-Pro as the disassembly front-end. IDA-Pro is a highly effective tool that can deal even with some complexities often found in malware. In addition, IDA-Pro can resolve many symbols needed for our analyses. The analyses are exactly the same for Windows and Linux (implemented in BinQ), c.f. Table 1.

Metric	Linux	Windows
# Files	970	410
# Unsafe Function Calls	9,424	1,964
# Complex Functions	3,583	3,195
# Malloc without Free	1,912	797
# Dangling Pointers	6,506	1,732
Sum of detected flaws	21,425	7,688
Ratio of flaws/files	22.1	18.6

Table 1: Binary Analysis Results.

We analyzed 970 binary files in the Debian Minimal distribution and noticed that many bad coding practices, according to our few metrics, are present. This may be an indication that Linux developers pay less attention to exactly these coding styles that we chose. Figure 3 illustrates our results for Linux. The x-axis represents the different files and the y-axis represents the number of flaws (bad coding practices according to our metrics) that we identified. Examples of files that have a distinctive amount of flaws (y-axis) in that image are:

- *for Unsafe Function Calls:*/bin/bash(437), /usr/bin/gpg(324), /usr/bin/ex (302), /bin/busybox (263), /usr/lib/libdb-4.4.so (210), /bin/netstat(183), /usr/lib/libdb-4.3.so(181), /usr/bin/tack(163), and /usr/lib/libdb-4.2.so(162).
- *for High Complexity:* /usr/bin/aptitude(200), /usr/bin/ex(160), /usr/lin/libdb-4.4.so(137), /usr/bin/perl(137), /usr/lib/libdb-4.3.so(106), and /usr/bin/gpg(105).
- *for Dangling Pointers:* /usr/lib/libkrb5.so.3(485), /lib/libsepol.so.1(243), /usr/bin/dpkg (214), /usr/bin/info (165), /bin/bash(151), and /bin/nano(144).
- *for Malloc without Free:* /usr/lib/libkrb5.so.3(193), /lib/libsepol.so.1(87), and /usr/bin/dpkg(83).

For Windows, we analyzed 410 binary files in the system32 directory. Figure 4 illustrates our results. Examples of files that have a distinctive amount of bad coding practices (y-axis) are:

- *for Unsafe Function Calls:* URTTemp/mscorwks.dll(538), infosoft.dll (188), URTTemp/msvcr71.dll (144), mfc42.dll (139), mfc42u.dll(139), and drmv2clt.dll(117).
- *for High Complexity:* URTTemp/mscorwks.dll(208), d3d9.dll(114), shell32.dll(91), and lsasrv.dell(76).
- *for Dangling Pointers:* URTTemp/mscorwks.dll(395), ntbackup.exe (217), URTTemp/msvcr71.dll (203), msvcrt.dll (182), hypertrm.dll(123), and infosoft.dll(97).
- *for Malloc without Free:* ntbackup.exe(151), URTTemp/msvcr71.dll(112), msvcrt(105), hypertrm.dll(81), and infosoft.dll(40).

It appears that the software quality, based on our few metrics, of Linux and Windows is poor. Both systems reveal a high number of bad software coding practices that are known to cause problems [2, 6]. However, it is possible that we chose metrics that are not part of good coding practices of neither Microsoft Windows or the Linux development community.

5. VERIFICATION

We have manually inspected random files in both Linux and Windows and judged from the disassembly whether our results are correct. In the case of Linux, we were sometimes able to find corresponding source code and verify that the problem was also inherent in the source. For instance, the following code snippet is part of /sbin/getty. Our *Malloc without Free* analysis reports one malloc without free. The corresponding source code contains only one malloc allocation in the following code:

```
...
 for (i=0; i < MAXDEF; i++) {
  if ((dp = defread(fp)) == (DEF *) NULL)
    break;
  if ((next =(DEF*) malloc((unsigned) sizeof(DEF))) ==
       (DEF *) NULL) {
    logerr(‘‘malloc() failed:defaults list truncated’’);
    break;
  }
  next->name = dp->name;
   next->value = dp->value;
   deflist[i] = next;
   debug(D_DEF, ‘‘deflist[%d]: name=(%s), value=(%s)’’,
      i, deflist[i]->name, deflist[i]->value);
  }
  deflist[i] = (DEF *) NULL;      /* terminate list */
  (void) defclose(fp);
  debug(D_DEF, ‘‘defbuild() successful’’);
  return(deflist);
 }
```

We can see that free() is not called (after the call to malloc) - at least not as part of this function, which is the requirement as defined by CERT [2]. In the following is an example assembly code snippet from tasklist.exe (Windows).

```
1005b4d je ...
1005b4f push DWORD PTR ss:[0xfffffffc + ebp]
1005b52 call PTR ds:[0x1001248]  <free>
1005b58 pop ecx
1005b59 jmp 0x1005b64
1005b5b mov ...
1005b64 xor eax, eax
1005b66 inc eax
1005b67 pop ebx
1005b68 pop ed
1005b69 pop esi
1005b6a leave
1005b6b ret
```

Our analysis correctly detected a *Dangling Pointer* flaw in the code above: the variable *0xfffffffc + ebp* is never set to 0 after free() is called. We expect that the false positive rate for the *Unsafe Function* analysis is zero, which is the same for the *Complex Function* analysis The other two analyses may have some false positives because these analyses use control flow information but no data-flow information or symbol evaluation. Such information would possibly increase the precision of these two metrics and is part of our future work.

6. RELATED WORK

GrammaTech [9] has demonstrated particularly broad capabilities and developed both source code and binary analysis tools. The source code analysis version is a high quality commercial static analysis tool and the binary analysis is specific to Windows x86 and is not openly available. These tools automate the detection of violations to numerous predefined rules and are useful for detailed analysis for software code for quality inspections, subtle bugs, and security violations. The BitBlaze [1] project at Berkeley is focused on binary analysis infrastructure and provides an openly available analysis capability for Linux x86 binaries. Other tools tools for binary analysis can be found at [7].

7. CONCLUSION AND FUTURE WORK

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. This denies users 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). In this paper, we attempted to measure simple quality features of binary software, namely Linux and Windows. Interestingly, our quality measurements do not show off either Windows or Linux favorably - but this may be even typical of OS implementations. The point is however that these metrics (and we expect many more metrics in the future) can be evaluated directly on binaries and therefore indirectly reflect properties of source code. Future work will focus on the improvement of our binary analysis capabilities.

8. REFERENCES

[1] BitBlaze. http://bitblaze.cs.berkeley.edu, 2009.

[2] CERT. Secure Coding Standards, 2007. https://www.securecoding.cert.org/confluence/.

[3] DATARESCUE. IDA - Interactive Disassembler, 2007. http://www.datarescue.com/.

[4] Edison Design Group. EDG front-end. http://www.edg.com.

[5] W. Li and S. Henry. Maintenance Metrics for the Object Oriented Paradigm. In *IEEE Proc. of the 1st Int. Software Metrics Symposium*, pages 52–60, May 1993.

[6] MITRE Corporation. Common Weakness Enumeration, 2007. http://cwe.mitre.org/.

[7] NIST. Binary Code Scanners. https://samate.nist.gov/index.php/Binary_Code_Scanners, 2009.

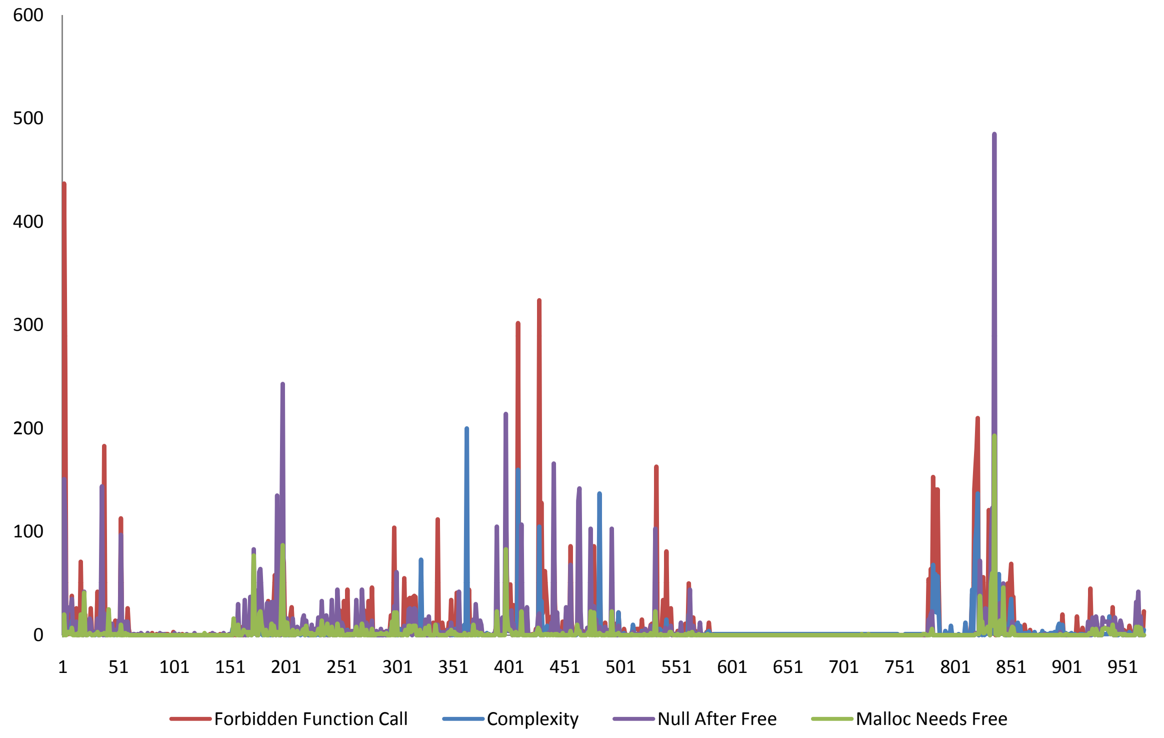


Figure 3: Debian Linux, parsed using IDA-Pro. X-Axis: Files - Y-Axis: # Flaws.

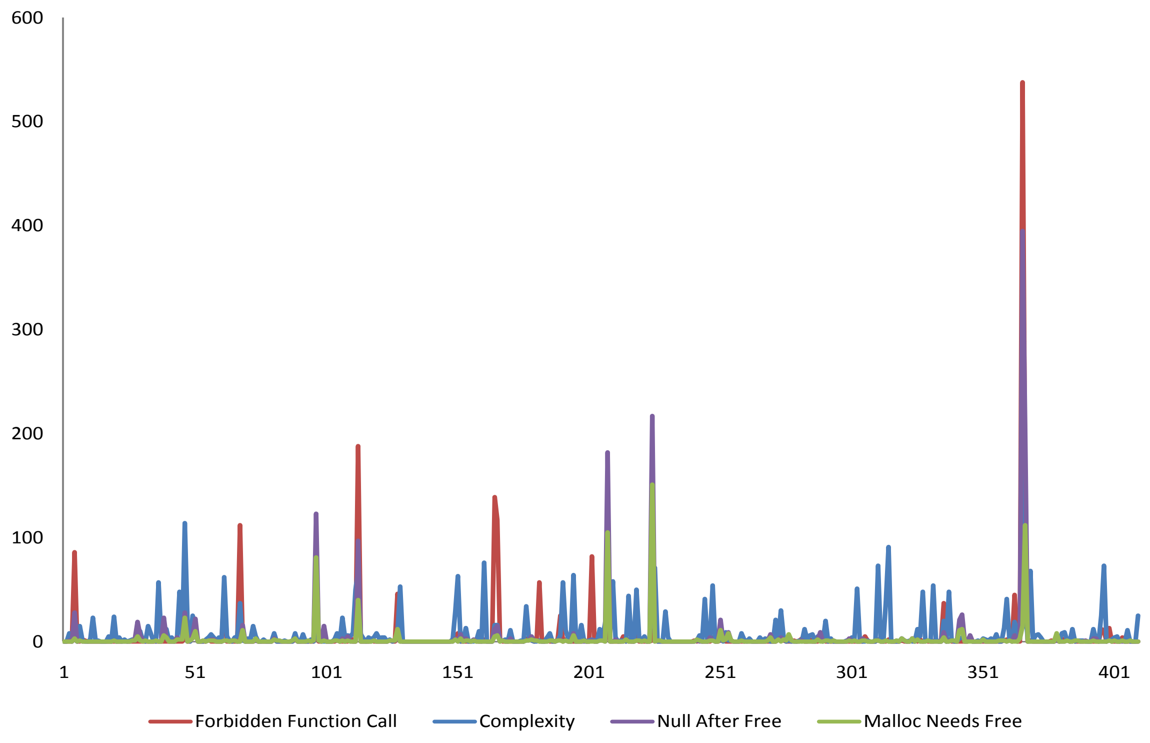


Figure 4: Windows, parsed using IDA-Pro. X-Axis: Files - Y-Axis: # Flaws.

[8] C. Rasmussen et al. Open Fortran Parser. http://fortran-parser.sourceforge.net/.

[9] Reps, T. and Balakrishnan, G. and Lim, J. and Teitelbaum, T. . A next-generation platform for analyzing executables. *Programming Languages and Systems*, 3780/2005:212–229, 2005.

[10] ROSE. Rose compiler, 2008. http://www.rosecompiler.org/.