Automatically Mining Program Build Information via Signature Matching

[Extended Abstract]

Charng-da Lu, Matthew D. Jones, Thomas R. Furlani
Center for Computational Research
SUNY at Buffalo
Buffalo, NY 14203
{charngda,jonesm,furlani}@ccr.buffalo.edu

ABSTRACT

Program build information, such as compilers and libraries used, is vitally important in an auditing and benchmarking framework for HPC systems. We have developed a tool to automatically extract this information using signature-based detection, a common strategy employed by anti-virus software to search for known patterns of data within the program binaries. We formulate the patterns from various "features" embedded in the program binaries, and the experiment shows that our tool can successfully identify many different compilers, libraries, and their versions.

Categories and Subject Descriptors

K.6.4 [**Management of Computing and Information Systems**]: System Management—*Management audit*; D.3.4 [**Programming Languages**]: Processors—*Compilers, code generation*

General Terms

Management

Keywords

Technology audit, program provenance, static binary analysis, ClamAV

1. INTRODUCTION

One important component in an auditing and benchmarking framework for HPC systems [1] is to be able to report the build information of program binaries. This is because the program performance depends heavily on the compilers, numerical libraries, and communication libraries. Moreover, as mentioned in [2], there is an increasing interest from funding bodies such as National Science Foundation in software and library usage on HPC systems they financed. This information can gauge how well the HPC resources are meeting their funding initiatives and scientific goals. From the perspectives of system administrators, the program build information can also help determine which compilers and libraries are rarely used and hence safe to retire.

However, in most HPC systems, program build information, if maintained at all, is recorded manually by system administrators. Over time, the sheer number of software/library packages of different versions, builds, and compilers of choice can grow exponentially and become too daunting and burdensome to document. For example, at our local center we have software packages built from 250 combinations of different compilers and numerical/MPI libraries. On larger systems such as Jaguar and Kraken at the Oak Ridge National Laboratory, the number can be as high as 738 [2].

In this paper, we present a signature-matching approach to automatically uncover the program build information. This approach is akin to the common strategy employed by anti-virus software to detect malware: search for a set of known signatures. We exploit the following "features" of program binaries and create signatures out of them:

- Compiler-specific code snippets.
- Compiler-specific meta data.
- Library code snippets.
- Symbol versioning.
- Checksums.

Our approach has several advantages. First, we only need to create, annotate, and maintain a database of signatures gathered from compilers and libraries, and we can then run the signature scanner over program binaries to derive their build information. Second, unlike the anti-virus industry where the malware code must be identified and extracted by experts, our signature collection process is almost mechanical and can be performed by non-experts. Third, our approach does not rely on symbolic information and thus can handle stripped program binaries.

Our implementation is based on the advanced pattern matching engine of ClamAV [3], an open-source anti-virus package. We choose ClamAV for its open-source nature, signature expressiveness and scanning speed.

2. PROGRAM BINARY CHARACTERISTICS

On most modern UNIX and its derivatives, the executable binaries (both programs and libraries) are stored in a standard object file format called the Executable and Linking

TeraGrid '11, July 18-21, 2011, Salt Lake City, Utah, USA.

Format (ELF) [4]. An ELF file can be divided into named "sections," each of which serves a specific function at compile time or runtime. There is a wealth of information embedded in these sections and we give a brief description below.

The first example is the so-called "processor dispatch" code inserted (unbeknownst to the developers) by certain x86 optimizing compilers, e.g. Intel and PGI. Its purpose is to detect the capabilities (e.g. SSE3, SSE4.x, AVX) of the CPU on which the program will run and re-route the execution path to the code chunks optimized for that capability. This detection is necessary to avoid potential "illegal instruction" type errors.

The second example is the `.comment` section in ELF files. It consists of null-terminated ASCII strings and is not loaded into memory during execution. Almost all compilers we examined fill this section with their unique brand strings and version numbers. Since this section is not purged by the GNU `strip` utility, we can mine it to obtain the compiler provenance.

The third example is statically-linked executables, which contain code chunks from libraries they linked to. We can obtain the library provenance of such executables by scanning for static libraries code. For dynamically-linked executables, the libraries can be easily identified by `ldd` utility and the MD5 checksums of dynamic libraries.

The last example is a new feature introduced to ELF file format called symbol versioning. This is a uniform self-annotation for specifying the versions of routines in dynamic libraries and the purpose is to allow better interoperability and easier management of dynamic libraries of different versions. Symbol versioning is used extensively in the GNU compiler collection (C, C++, Fortran, and OpenMP runtime libraries), Myrinet libraries, and OpenFabrics / InfiniBand Verbs libraries. It is possible to recognize these libraries and their versions by simply examining their symbol version tables.

3. IMPLEMENTATION

Our impementation is based on the open-source anti-virus package ClamAV [3]. It comprises two tools: a signature generator and a signature scanner. The signature generator takes ELF files and automatically outputs ClamAV-formatted signature files. The signature scanner takes as input the signature files and the executable binaries and outputs all possible matches.

ClamAV has several different signature formats and our implementation uses the Basic (plain hexadecimal strings) and the RegEx (regular expression) formats because they can be generated automatically. ClamAV's pattern matching algorithms are Wu-Manber and Aho-Corasick, which can also be found in UNIX `grep` utility. The difference is ClamAV's implementation is highly optimized for seeking a large set of signatures (tens of thousands) in a text.

4. EVALUATION

We evaluate our approach with both toy programs and real-world HPC applications from our center. We compile toy programs with a variety of compilers to test the effectiveness of source compiler identification. We use the existing HPC applications to assess both the compiler/library recognition and ClamAV's scanning performance.

We use fourteen 64-bit compilers on x86 Linux in our tests with toy programs, and our scanner is able to identify all but one (Clang) compilers. Since the output shows all possible matches, we found that GCC is often in the matches as well, even though it is not the compiler used. This is because many C compilers strive to be compatible with GCC, so they also use GCC's code snippets. The results also show that we can find the versions of the compilers used, for example, the output from a PGI-compiled code:

```
Matches:
(58 times, 346766 bytes) PGI Fortran Compiler 11.x
(48 times, 56833 bytes)  PGI Fortran Compiler 8.x
(45 times, 118288 bytes) PGI Fortran Compiler 10.x
(42 times, 49895 bytes)  PGI Fortran Compiler 7.x
(32 times, 82808 bytes)  PGI Compiler Suite 11.x
(29 times, 57166 bytes)  PGI Compiler Suite 7.x
....
(2 times, 200 bytes) GCC 4.4.3
```

Such an output is typical, since many compilers reuse a significant amount of code across each release.

We applied the scanner to a variety of real-world HPC applications from our center and a Cray XT5: Amber Charmm, CPMD, GAMESS, Lammps, NAMD, NWChem, and PWscf. We gather signatures from numerical and MPI libraries which we know have been linked statically in these application builds. The signature database has 100K signatures (one signature per library routine). Generating this database is very fast: The largest library we have ever encountered is 210 MB (`libmkl_core.a` in Intel MKL 10.3.1) and it takes 28 seconds (78% of this time is spent in I/O) to extract its signatures. We also apply a compression scheme so each signature is no longer than 256 bytes long. Our results show that the scanner can correctly identify all used libraries. The scanning time t (in seconds) can be best described by the linear regressions $t = -1.11 + 7.23x$ (2.5 GHz Intel Xeon L5420 "Harpertown") and $t = -5.44 + 6.98x$ (2.8 GHz X5560 "Nehalem" node) where x is the application code size in MB, and the scanner's peak memory usage is 195 MB.

5. CONCLUSIONS

Compilers and libraries provenance reporting is crucial in an auditing and benchmarking framework for HPC systems. In this paper we present a simple and effective way to mine this information via signature matching. Our tests show correct identification of compilers and libraries and excellent scanning speed on real-world HPC applications.

Acknowledgments

This work is supported by the National Science Foundation under award number OCI 1025159.

6. REFERENCES

[1] T. R. Furlani and *et al.* Performance metrics and auditing framework using applications kernels for high performance computer systems. In preparation.

[2] B. Hadri, M. Fahey, and N. Jones. Identifying software usage at HPC centers with the automatic library tracking database. In *TeraGrid Conference Proceedings*, 2010.

[3] T. Kojm. http://www.clamav.net.

[4] M. Wilding and D. Behman. *Self-service Linux: Mastering the art of problem determination.* Prentice Hall, 2005.