

# Automatically Mining Program Build Information via Signature Matching

Charng-Da Lu  
Buffalo, NY 14203

## 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.

## 1 Introduction

One important component in an auditing and benchmarking framework for HPC systems 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. For example, the SPEC CPU 2000 Run and Reporting Rules [2] contain meticulous guidelines on the reporting of the compiler of choice, compilation flags, allowed and forbidden compiler tuning, libraries, data type sizes, etc.

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 [13].

In addition, there is no standard format of documenting program build information. Many HPC systems use Modules [3] or SoftEnv [4] to manage software pack-

ages, and a common naming scheme is to incorporate the compiler name (as a suffix) in the package name. There is usually additional textual description to indicate build information, such as compiler version, debug/optimization/profiling build, and so on. Mining these free-form texts, however, requires the understanding of each HPC site's software environment and documentation style and is not generally applicable.

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 [11], an open-source anti-virus package. We choose ClamAV for its open-source nature, signature expressiveness and scanning speed.

The remainder of this paper begins by describing the features in the program binaries. Section 3-4 provide the implementation details and experimental results. We then

discuss potential improvement and related work in § 5-6, followed by a conclusion in §7.

## 2 Program Binary Characteristics

On most modern UNIX and UNIX-related systems, the executable binaries (programs and libraries) are stored in a standard object file format called the Executable and Linking Format (ELF) [5, 6]. An ELF file can be divided into named "sections," each of which serves a specific function at compile time or runtime. The sections relevant to our work are:

- `.text` section contains the executable machine code and is the main source for our signature identification.
- `.comment` section contains compiler and linker specific version control information. More on this in §2.2.
- `.dynamic` section holds dynamic linking information, including file names of dependent dynamic libraries, and pointers to symbol version tables and relocation tables.
- `.rel.text` and `.rela.text` sections consist of relocation tables associated with the corresponding `.text` sections. More details in §3.2.
- `.gnu.version_d` section comprises the version definition table. More on this in §2.4.

There is a wealth of information embedded in these sections, and in the following we explain these characteristics in detail.

### 2.1 Compiler-Specific Code Snippets

It is not news that certain popular compilers on the Intel x86 platform insert extra code snippets unbeknownst to the developers [7]. We will illustrate with three examples.

The first example is the so-called "processor dispatch" employed by certain optimizing compilers. As the x86 architecture evolves with the addition of new capabilities and new instructions such as Streaming SIMD Extensions (SSE) and Advanced Vector eXtensions (AVX), an optimizing compiler will produce machine code tuned for each capability. Since the new instructions are not recognized by older generations of x86 processors, to avoid "illegal instruction" errors and to re-route the execution path to the suitable code blocks, an extra code snippet is inserted to perform this task.

Both Intel and PGI compilers, when invoked with optimization flags enabled (and `-O2` is used implicitly), insert the processor dispatch code which is executed before the application's `main` function. These code snippets invariably use the `cpuid` instruction to obtain processor feature flags. For example, the core processor dispatch routine used by the Intel compiler is called `__intel_cpu_indicator_init`. It initializes an internal variable called `__intel_cpu_indicator` to different values based on the processor on which the program is running [7]. This information is later used to either abort program execution immediately, with an error like "This program was not built to run on the processor in your system," or execute different code blocks (tuned for different generations of SSE instructions) in Intel's optimized C library routines such as `memcpy` and `strcmp`.

A second instance of compiler-inserted code is to enable or disable certain floating-point unit (FPU) features. For example, when GCC is invoked with `-ffast-math` or `-funsafe-math-optimizations` optimization flags, it inserts code to turn on the Flush-To-Zero (FTZ) mode and the Denormals-Are-Zero (DAZ) mode in the x86 control register `MXCSR`. When these modes are on, the FPU bypasses IEEE 754 standards and treats denormal numbers, i.e. values extremely close to zero, as zeros. This optimization trades off accuracy for speed [8]. The GNU C Compiler, GCC, also accepts `-mpc{32|64|80}` flags, which are used to set the legacy x87 FPU precision/rounding mode. Again, GCC uses a special prolog code to configure the FPU to the requested mode.

A third instance of compiler-inserted code is to initialize user's data. For example, one of the C++ language features requires that static objects must be initialized, i.e. their constructors must be called, before program startup [9]. To implement this, the C++ compiler emits a special ELF section called `.ctors`, which is an array of pointers to static objects' constructors, and inserts a prolog code snippet which sweeps through the `.ctors` section before running the application's `main` function.

### 2.2 Compiler-Specific Meta Data

ELF files have an optional section called `.comment` which consists of a sequence of null-terminated ASCII strings. This section is not loaded into memory during execution and its primary use is a placeholder for version control software such as CVS or SVN to store control keyword information. In practice, most compilers we examined will also fill this section with strings which are unique enough to differentiate the compilers and the ver-

sions (see §4.1). The compiler adds string data by using the `.ident` assembler directive when generating the assembly code, and then the assembler pools these strings and saves them into the `.comment` section. Unlike the debugging and symbolic information embedded in other ELF sections, the `.comment` section is not removed by the GNU `strip` utility, so we can mine it to obtain the compiler provenance.

For example, using the GNU `readelf` tool with command-line option `-p .comment` on GCC-compiled programs could have the following output:

```
GCC: (GNU) 4.1.2 20080704 (Red Hat 4.1.2-50)
```

## 2.3 Library Code Snippets

If a program calls library functions, the linker will bind the functions to libraries to create the executable. The linking mode is either static or dynamic. In the former, the linker extracts the code of called functions from libraries, which are simply archives of ELF files, and performs the relocation (see §3.2) to merge the user's code and the library functions code into a single executable. In the latter, the linker does not use any code from the libraries, but instead creates proxy/stub code which can locate the entry point of each called library function at runtime.

Static linking, although it has drawbacks such as code duplication and is no longer the default mode of linking on most platforms, is still used in cases where dynamic linking is problematic. For example, unlike C, C++ and Fortran do not have an agreed API and ABI (application binary interface), so not only object files created by different C++/Fortran compilers can seldom be linked together, object files created by different versions of the same compiler are not guaranteed to interoperate either [14, 15]. For this reason, Fortran compilers in particular, tend to use static linking. It is also not uncommon for independent software vendors (ISVs) to ship only statically linked binaries to avoid portability and library dependency issues.

On some platforms where the operating system is designed to be simple and efficient, e.g. Cray XT's Cata-mount and IBM Blue Gene/L's Compute Node Kernel (CNK), dynamic linking is usually not an option and static linking has to be used [17].

A third case for static linking is the aforementioned compiler-specific code snippets. They exist as object files or libraries and are almost always statically linked.

For all of above reasons, library code snippets are the most important source of signatures in our program build discovery tool.

## 2.4 Symbol Versioning

Some dynamic libraries are self-annotated with version information in a uniform format, and we use this information to identify both the library and its version.

As mentioned in §2.3, dynamic linking has the issue of interoperability. Historically, this was partly solved by having unique file names for the dynamic libraries. The file names usually incorporate major and minor release numbers, such as `lib<name>.so.<major>.<minor>`. The linker will then record the exact file names in the resulting binaries' `.dynamic` section. In 1995 Sun introduced a new and fine-grained versioning mechanism in Solaris 2.5, which the GNU/Linux community soon adopted [12]. In this scheme, each function name and symbol can be associated with a version, and at the library level, a chain of version compatibility can be specified. *The version of the library is then the highest version in the version chain.*

As an example, in the GNU C runtime library (`glibc`) source tree, one can find version definition scripts containing the following

```
libc {
    GLIBC_2.0 {
        malloc;
        free;
        ...
    }
    ...
    GLIBC_2.10 {
        malloc_info;
    }
}

libc {
    GLIBC_2.0
    GLIBC_2.1
    ...
    GLIBC_2.10
    ...
}
```

The left-hand side specifies that `malloc` and `free` are versioned `GLIBC_2.0` and `malloc_info` `GLIBC_2.10`. The right-hand side indicates `GLIBC_2.10` is compatible with `GLIBC_2.1`, which is compatible with `GLIBC_2.0`. All of the versioning data are encoded in the `.gnu.version_d` section (`d` for definition) of dynamic libraries when they are built. When a user program is compiled and linked, a version-aware linker obtains versions of called functions from the dynamic libraries and stores them in the resulting binaries' `.gnu.version_r` section (`r` for reference). At runtime, the program loader-linker `ld.so` first examines whether all version references in the user's program binary can be satisfied or not, and determines to either abort or continue.

Symbol versioning is used extensively in the GNU compiler collection (C, C++, Fortran, and OpenMP runtime libraries), Myrinet MX/DAPL libraries, and OpenFabrics/InfiniBand Verbs libraries. All of these instances adopt the same version naming scheme: a unique label,

e.g. `GLIBC`, `GLIBCXX`, or `MX`, followed by an underscore and the version. Hence, our tool can recognize them using a hard-coded list of labels and obtain their version by traversing the version chain.

## 2.5 Checksums

Most dynamic libraries are less sophisticated and do not use symbol versioning. Therefore, to recognize them, we resort to the traditional approach of checksums. `Md5sum` is a commonly used open-source utility to produce and verify the MD5 checksum of a file, but it is file-structure agnostic and fails to characterize ELF dynamic libraries on platforms (e.g. Red Hat Enterprise Linux) where the prelinking/prebinding technology [18] is used. Prelinking is intended to speed up the runtime loading and linking of dynamic libraries when a program binary is launched. To achieve this, a daemon process will periodically update the dynamic libraries' relocation table. The side effect of prelinking is MD5 checksum mismatch, as part of the file content has been changed. To defeat this effect, we calculate the MD5 checksum over the `.text` section only for ELF files.

## 3 Implementation

Our implementation is based on the pattern matching engine of the open-source anti-virus package ClamAV [11], with additional code to support symbol versioning. The implementation comprises two tools: a signature generator and a signature scanner. The signature generator parses ELF files and outputs ClamAV-formatted signature files. The signature scanner takes as input the signature files and the user's program binary and outputs all possible matches. In the following, we discuss ClamAV's signature formats and matching algorithms and how we leverage ClamAV in our implementation.

### 3.1 ClamAV Design

ClamAV signatures can be classified as one of the following types, in the order of increasing complexity and power: **MD5**, **basic**, **regular expression (regex)**, **logical**, and **bytecode**. Our implementation makes use of the first three types because they can be generated automatically (see §3.2).

A basic signature is a hexadecimal string. ClamAV's scanning engine handles this type of signature with a modified version of the classical Boyer-Moore string searching algorithm called Wu-Manber. A regex signature is a basic signature with wildcards. Our implementation use

two kinds of wildcards extensively: `??` (to match any byte) and `{n}` (to match any consecutive  $n$  bytes). ClamAV's scanning engine handles regex signatures with the Aho-Corasick (AC) string searching algorithm, which can match multiple strings concurrently at the cost of consuming more memory. The AC algorithm starts with a preprocessing phase: Take a set of wildcard-free strings to create a finite automaton. The scanning phase is simply a series of state transitions in this finite automaton. ClamAV utilizes the AC algorithm as follows: Every regex signature is broken into basic signatures (separated by wildcards), and a single finite automaton (implemented as a two-level 256-way "trie" data structure) is created from all of these basic signatures. If all wildcard-free parts of a regex signature are matched, ClamAV checks whether the order and the gaps between the parts satisfy the specified wildcards.

For completeness we briefly mention the remaining two signature types. We do not use them because we do not yet find automatic ways to create them. Logical signatures allow combining of multiple regex signatures using logical and arithmetic operators. Bytecode signatures further extend logical signatures and offer the maximal flexibility. Bytecode signatures are actually ClamAV plug-ins compiled from C programs into LLVM bytecodes, and hence allow arbitrary algorithmic detections of patterns.

### 3.2 Signature Generator

For dynamic libraries (`.so` files), the signature generator computes the MD5 checksums over their `.text` sections and outputs the ClamAV-conformant MD5 signature files.

Compiler-specific code snippets and static library code reside in ELF `.o` (object) and `.a` (library archive) files. In the following discussions we only focus on `.o` file handling because an `.a` file is just an archive of multiple `.o` files. Our signature generator extracts `.text` sections from `.o` files, and outputs, for each `.text` section, a basic or regex signature of 16-255 bytes length (excluding the wildcards.) We describe this process in depth as follows.

First, *a signature is not just bytes from the `.text` section verbatim*. When a source file is compiled into an `.o` file, the addresses of unresolved function names and symbols in this `.o` file are unknown and have to be left empty. It is during the linking phase that these addresses are resolved and assigned by the linker. This process is called *relocation* [10]. To facilitate the relocation, the compiler emits one relocation table for each `.text` section. Each entry of a relocation table specifies the symbol name to be resolved, the offset into the `.text` section which con-

tains the address to be assigned, and the relocation type. When we create a signature from the bytes of a `.text` section, we have to *mask the bytes which are reserved for addresses yet to be computed*. To illustrate, suppose that we compile the following source code into an `.o` file:

```
#include <stdlib.h>
void foo() {
    char *buf = malloc(10);
}
```

On x86, the disassembly of the generated `.o` file would be (using the GNU `objdump` utility):

```
000000 <foo>:
0: 55                push    %rbp
1: 48 89 e5          mov     %rsp,%rbp
4: 48 83 ec 10       sub     $0x10,%rsp
8: bf 0a 00 00 00    mov     $0xa,%edi
d: e8 00 00 00 00    callq  12 <foo+0x12>
12: 48 89 45 f8       mov     %rax,-0x8(%rbp)
16: c9                leaveq  %rax
17: c3                retq
```

and the corresponding relocation table is:

OFFSET	TYPE	VALUE
00000e	R_X86_64_PC32	malloc+0xffffffff ffffffffc

Together, the above examples illustrate that the target of the `callq` instruction should be the address of a function named “`malloc`”, and the address should fill the 4 bytes (as specified by the `R_X86_64_PC32` relocation type) starting at offset `0xe` (the boxed `00`’s). So if `foo`, as a library function, is used to create a user program binary, the linker will take the byte stream `55 48 89 e5 ... c9 c3` and fill the bytes at offset `0xe` through `0xe+3` with the actual address of `malloc`. Thus, to identify `foo`, we create a ClamAV regex signature as:

```
55 48 89 e5 48 83 ec 10 bf 0a 00 00
00 e8 ?? ?? ?? 48 89 45 f8 c9 c3
```

The second consideration is the signature size. As will be seen in §4.2 a `.text` section can be as big as four megabytes. Using the entire `.text` section could lead to long preprocessing time and large disk/memory storage space. Therefore, we impose an upper limit on the signature size to be 255 bytes. We think 255 is a reasonable size, as there are  $256^{255}$  possible distinct 256-byte streams, which is large enough to have few collisions/false positives. For a `.text` section of  $n > 256$  bytes, we use the tailing  $255/3=85$  bytes  $x_1x_2 \dots x_{85}$  of the first third portion, the tailing 85 bytes  $y_1y_2 \dots y_{85}$  of the middle third, and the tailing 85 bytes  $z_1z_2 \dots z_{85}$  of the last middle third, and form a regex signature as:

$$x_1x_2 \dots x_{85} \{l\} y_1y_2 \dots y_{85} \{m\} z_1z_2 \dots z_{85}$$

where  $l = \lfloor n/3 \rfloor - 85$  and  $m = l + (n \% 3)$ . We also ignore `.text` sections which are shorter than 16 bytes. This cut-off is chosen because the size of an x86 instruction varies between 1 and 16 bytes, and since we do not decode the bytes back to x86 instructions, we do not know the instruction boundaries and have to make a conservative assumption. Besides, signatures that are too short could result in many false positives.

The third consideration is an `.o` file could contain more than one `.text` section. This happens in GNU Fortran’s static library, which is created with the `-ffunction-sections` compiler flag. This flag instructs the compiler to put each function in its own `.text` section instead of all functions from the same source file in one single `.text` section. So for a Fortran function, say `foo`, the compiler creates a section named `.text.foo` which consists of `foo`’s code only<sup>1</sup>. In such a situation, our tool emits one signature for one such `.text` section.

### 3.3 Signature Scanner

The signature database is organized as a collection of signature files, each of which contain signatures from a specific compiler/library, e.g. Intel Fortran compiler, Intel MKL, MVAPICH, etc. Each signature file is annotated manually to indicate the package name and version. The scanner takes as input this database and the user’s program binary and outputs all possible matches. For dynamic library identification, it uses the `ldd` command to obtain the library pathnames. It then extracts their symbol versioning data (if there is any) and compares against a list of known labels, as explained in §2.4. For those without symbol versioning, the scanner checks their MD5 checksums against those in the database.

For compiler and static library identification, the scanner loads the program binary’s `.text` and `.comment` sections (compiler meta-data are treated as basic signatures) and runs them through the ClamAV matching engine. By default ClamAV stops as soon as it spots a match, so to find all matches, we modify it by repeatedly zeroing out the matched area and rerunning the engine, until no match can be found.

<sup>1</sup>This optimization reduces the size of statically linked program binaries because it eliminates dead code, i.e. functions which are unused but included nevertheless because they are in the same source files as the used functions.

## 4 Evaluation

We evaluate our approach with both toy programs and real-world HPC software packages from two HPC sites. We compile toy programs with a variety of compilers to test the effectiveness of source compiler identification. We use the existing HPC software packages to assess not only the compiler and library recognition but also ClamAV’s scanning performance.

### 4.1 Compiler Identification

We examine fourteen compilers on the x86-64 Linux platform and we summarize our findings in Table 1. We locate the compiler-specific code snippets by enabling the verbosity flag in building the toy programs. This flag is supported by all compilers and it can display exactly where and which `.a` and `.o` files are used in the compilation process. The toy programs we constructed, e.g. ”Hello, World” and matrix multiplication, are short and use only basic language features and APIs, so they can highlight the usefulness of our approach. All test cases are compiled with each compilers’ default settings.

As an example, the ”Hello, World” program compiled with Intel compiler 12.0 yields the following output from our scanner. It gives the number of matches and total size of matches against each signature file:

```
(3 times, 6992 bytes) Intel Compiler Suite 12.0
(2 times, 200 bytes) GCC 4.4.3
```

We have the following observations. 1. Many compilers strive to be compatible with the GNU development tools and runtime environment, so they also use GNU’s code snippets. Therefore, GCC becomes a common denominator and is ubiquitous in the scanning results. The above output is typical: The Intel compiler locates the system’s default GCC installation (version 4.4.3 in this case) and uses its `crtbegin.o` and `crtend.o` in the compilation. These two `.o` files handle the `.ctors` section as discussed in §2.1.

2. As opposed to C, the Fortran compiler space is very fragmented, with each compiler having its own implementation of language intrinsics and extensions. Hence, we can spot a Fortran compiler by just examining the runtime library code used. The same is true for OpenMP and UPC.

3. It is possible to recognize different versions of the same compiler. To demonstrate, we wrote a simple Fortran program which calls the `matmul` intrinsic to perform matrix multiplications and compiled it with PGI 11.0. The result is as follows:

```
(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
```

Compiler	Note	Version	Meta Data	Code Snippet Source
Absoft	F,O	11.1		liba*.a
Clang	C,L	2.8		
Cray		7.1, 7.2	V	libsup.a, libf*.a, libcray*.a
G95	F,G	0.93	V	libf95.a
GNU	G	4.1, 4.4, 4.5	V	crt*.o, libgcc*.a
Intel		9.x thru 12.0	I	libirc*.a, libf- core*.a
Lahey- Fujitsu	F	8.1	I	fj*.o, libfj*.a
LLVM- GCC	G,L	2.8	V	
NAG	F,†	5.2		libf*.a
Open64	O,‡	4.2	V	libopen64*.a, libf*.a
PathScale	O,‡	3.2, 4.0	V	lib*crt.a, lib- path*.a
PCC	C	0.99	V	crt*.o, libpcc*.a
PGI		6.x thru 11.x	V	libpgc.a, libpgf*.a, f90*.o, pgf*.o
Sun Studio		12.x	V	crt*.o, libc_supp.a, libf*.a

Table 1: Compiler identification. C: C/C++ compiler only. F: Fortran compiler only. G: uses GNU codebase. I: has unique meta data. L: uses LLVM codebase. O: uses Open64 codebase. V: meta data have both brand string and version number. †: is actually a Fortran-to-C converter with GCC as backend. ‡: inserts FTZ/DAZ-enabling prolog code (see §2.1) but this code is not in any `.a/.o` files so we manually produce its signature.

Library	Version (Compiler)	Code Snippet Source	Mean and StdDev .text size in KB
ACML	4.4.0 (I,P)	libacml*.a	11.1, 70.8
Cray LibSci	10.4.0 (G,I,P)	libsci*.a	3.4, 4.9
Intel MKL	8.0, 8.1, 9.1	libmkl*.a	4.6, 9.0
	10.x	libmkl_core.a	4.2, 16.6
Cray MPI	3.5.1 (G,I,P)	libmpich*.a	1.3, 2.6
MPICH	1.2.7mx (G,I)	libmpich.a	1.2, 2.7
MVAPICH2	1.4, 1.5 (I)	libmpich.a	2.6, 4.8

Table 2: Library identification. G: GNU. I: Intel. P: PGI.

```
(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
```

The matches include both the Fortran runtime library and compiler-specific code snippets, which are shared by C/C++ and Fortran compilers. The result also implies that PGI reuses a significant amount of code across each release. We scrutinized the code snippets which matched both versions 7.x and 11.x and found their functionality includes memory operations (allocate, copy, zero, set), I/O setup (open, close), command-line argc/argv handling, etc.

4. Compilers which share codebase are not easily distinguishable. Examples include Open64 and PathScale, GNU and LLVM-GCC, etc. In these cases, only the compiler-specific meta data can tell them apart, and Clang is thus far the only compiler which defies our inference efforts.

## 4.2 Library Identification

We applied the scanner to a subset of HPC applications (Amber [20], Charmm [21], CPMD [22], GAMESS [23], LAMMPS [24], NAMD [25], NWChem [26], PWscf [27]) from two HPC sites (a 3456-core Intel-based commodity PC cluster at our center and a 672-core Cray XT5m at Indiana University). We gathered signatures from numerical and MPI libraries which we know have been linked statically in the application builds. The libraries and the size of their constituent `.o` files are summarized in Table 2. Numerical libraries tend to have more `.o` files and larger code size per `.o` file. The explanation is various processor-specialization codes and aggressive loop unrolling. For example, ACML 4.4.0-ift64’s `libacml.a` has 4.5K `.o` files, with the largest (4.1 MB code size) being an AMD-K8-tuned complex matrix multiplication (zgemm) kernel, and Intel MKL 10.3.1’s `libmkl_core.a` has 44K `.o`’s, with the largest (1.4 MB) being an Intel-Nehalem-optimized batched forward discrete Fourier transform code.

For the test we create a signature database exclusively from the aforementioned libraries. It has 100K signatures and the predominant signature type is regex. The 21 HPC application binaries under test have a mean code size of 13.3 MB and the largest is NWChem 6.0 on Cray (39.4 MB, mainly due to static linking, as in §2.3). We build the (single-threaded) scanner with Intel compiler 12.0 and we run the scan on a 2.5 GHz Intel Xeon L5420 “Harper-town” node and a 2.8 GHz X5560 “Nehalem” node. The 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  $t = -1.11 + 7.23x$  (Harper-town) and  $t = -5.44 + 6.98x$  (Nehalem) where  $x$  is the code size in MB, and the peak memory usage is 195 MB.

## 5 Discussion

Our methodology of identifying the source compiler depends on the idiosyncrasies of the x86 platform and compilers. We also explored the two major compilers, GCC and IBM XL, on the PowerPC platform, and did not find discernible compiler-specific code snippets. IBM XL compilers do inscribe their brand strings in the `.comment` section, but in general, content in `.comment` section is subject to tampering. For example, the following line in a C program:

```
__asm__(".ident \"foo\"");
```

will emit “foo” to the `.comment` section. This makes `.comment` section a less reliable source of compiler provenance from a general perspective of software forensics.

Another issue is that a compiler inserts its characteristic prolog code only when it is compiling the source file which contains the `main` function. So if different source files are compiled with different compilers, the resulting program binary could lack the compiler-specific code snippets one would expect. In addition, in Intel compiler’s case, it does not insert processor-dispatch code if the optimization is turned off either explicitly (with `-O0`) or implicitly (e.g. with `-g`).

Our approach cannot discover the compilation flags used in the program build process. Some compilers offer a switch to record the command-line options inside either `.comment` or other sections. For example, Intel has `-sox`, GCC has `-frecord-gcc-switches` (recorded in `.GCC.command.line` section), and Open64/PathScale and Absoft do it by default. We expect this self-annotation feature to be more widely embraced by compiler developers, as they move toward better compatibility with GCC, and used by HPC programmers, as it greatly aids debugging and performance analysis.

## 6 Related Work

ALTD [13] is an effort to track software and library usage at HPC sites. It takes a proactive approach by intercepting and recording every invocation of the linker and the job scheduler. Our work is complementary in that it performs post-mortem analysis and works on systems without ALTD.

The work by Rosenblum *et al* [16] is the first attempt to infer the compiler provenance. They used sophisticated machine learning by modeling and classifying the code byte stream as a linear chain Conditional Random Field. As in most supervised learning systems, a lengthy training phase is required. The resulting system can then infer the source compiler with a probability. Their approach has several drawbacks, which our method addresses: They focus solely on executable code and ignores other parts of ELF files, the preprocessing/training phase, albeit one-time, is slow and complex, the model parameters cannot be updated incrementally with ease when a new compiler is added, and it is unclear if their model can discern the nuances among different versions of the same compiler.

Kim’s approach [19] is closest to ours in spirit, but it misses the key feature in our implementation: the relocation table. It produces a signature by copying the first 25 bytes of a library function code *verbatim*. With such a short signature and lack of relocation information, his tool has very limited success in identifying library code snippets.

## 7 Conclusion

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. We also demonstrate that building and updating a signature database is straightforward and needs no expert knowledge. Finally, our tests show excellent scanning speed even on very large program binaries.

## Acknowledgments

This work is supported by the National Science Foundation under award number OCI 1025159. We would like to thank Gregor von Laszewski for providing access to FutureGrid computing resources.

## References

- [1] T. R. Furlani *et al.*, “Performance metrics and auditing framework using applications kernels for high performance computer systems.” In preparation.
- [2] <http://www.spec.org>
- [3] <http://modules.sf.net>
- [4] <http://www.teragrid.org/userinfo/softenv/>
- [5] M. Wilding and D. Behman, “Self-service Linux: Mastering the art of problem determination.” Prentice Hall, 2005.
- [6] “System V application binary interface - AMD64 architecture processor supplement.” <http://www.x86-64.org/documentation/>
- [7] A. Fog, Chapter 13 of “Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms.” <http://www.agner.org/optimize/>
- [8] I. Dooley and L. Kale, “Quantifying the interference caused by subnormal floating-point values.” *The Workshop on Operating System Interference in High Performance Applications (OSIHPA)*, 2005.
- [9] §8.5 of “Working Draft of Standard for Programming Language C++, Document No. N1905.” <http://www.open-std.org>
- [10] J. R. Levine, “Linkers and loaders.” Morgan Kaufmann, 1999.
- [11] T. Kojm, <http://www.clamav.net>
- [12] D. J. Brown and K. Runge, “Library interface versioning in Solaris and Linux.” *The 4th Annual Linux Showcase (ALS) & Conference*, 2000.
- [13] B. Hadri, M. Fahey, and N. Jones, “Identifying software usage at HPC centers with the automatic library tracking database.” *Proceedings of the 2010 TeraGrid Conference*.
- [14] N. Sidwell, “A common vendor ABI for C++ – GCC’s why, what and not.” *Proceedings of the 2003 ACCU Conference*.
- [15] <http://gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html>
- [16] N. Rosenblum, B. Miller, and X. Zhu, “Extracting compiler provenance from program binaries.” *The workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2010.
- [17] G. Johansen and B. Mauzy, “Cray XT programming environment’s implementation of dynamic shared libraries.” *Cray User Group (CUG) Conference*, 2009.
- [18] J. Jelinek, <http://people.redhat.com/jakub/prelink.pdf>
- [19] J. S. Kim, “Recovering debugging symbols from stripped static compiled binaries.” *Hakin9 Magazine*, June 2009. <http://0xbeefc0de.org/papers/>
- [20] D. A. Case *et al.*, “The Amber biomolecular simulation programs.” *J. Comp. Chem.* v 26, 1668-1688 (2005).
- [21] B. R. Brooks *et al.*, “CHARMM: The biomolecular simulation program.” *J. Comp. Chem.* v 30, 1545-1615 (2009).
- [22] <http://www.cpmc.org>
- [23] M. W. Schmidt *et al.*, “General atomic and molecular electronic structure system.” *J. Comp. Chem.* v 14, 1347-1363 (1993).
- [24] S. J. Plimpton, “Fast parallel algorithms for short-range molecular dynamics.” *J. Comp. Phys.* v 117, 1-19 (1995).
- [25] J. C. Phillips *et al.*, “Scalable molecular dynamics with NAMD.” *J. Comp. Chem.* v 26, 1781-1802 (2005).
- [26] M. Valiev *et al.*, “NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations.” *Comput. Phys. Commun.* v 181, 1477 (2010).
- [27] P. Giannozzi *et al.*, <http://www.quantum-espresso.org>