Run-time testing of LSB Applications

Stuart Anderson
Free Standards Group
anderson@freestandards.org

Matt Elder
University of South Caroilina
happymutant@sc.rr.com

Abstract

The dynamic application test tool is capable of checking API usage at run-time. The LSB defines only a subset of all possible parameter values to be valid. This tool is capable of checking these value while the application is running.

This paper will explain how this tool works, and highlight some of the more interesting implementation details such as how we managed to generate most of the code automatically, based on the interface descriptions contained in the LSB database.

Results to date will be presented, along with future plans and possible uses for this tool.

1 Introduction

The Linux Standard Base (LSB) Project began in 1998, when the Linux community came together and decided to take action to prevent GNU/Linux based operating systems from fragmenting in the same way UNIX operating systems did in the 1980s and 1990s. The LSB defines the Application Binary Interface (ABI) for the core part of a GNU/Linux system. As an ABI, the LSB defines the interface between the operating system and the applications. A complete set of tests for an ABI must be capable of measuring the interface from both sides.

Almost from the beginning, testing has been a cornerstone of the project. The LSB was originally organized around 3 components: the written specification, a sample implementation, and the test suites. The written specification is the ultimate definition of the LSB. Both the sample implementation, and the test suites yield to the authority of the written specification.

The sample implementation (SI) is a minimal subset of a GNU/Linux system that provides a runtime that implements the LSB, and as little else as possible. The SI is neither intended to be a minimal distribution, nor the basis for a distribution. Instead, it is used as both a proof of concept and a testing tool. Applications which are seeking certification are required to prove they execute correctly using the SI and two other distributions. The SI is also used to validate the runtime test suites.

The third component is testing. One of the things that strengthens the LSB is its ability to measure, and thus prove, conformance to the standard. Testing is achieved with an array of different test suites, each of which measures a different aspect of the specification.

LSB Runtime

- `cmdchk`

  This test suite is a simple existence test that ensures the required LSB commands and utilities are found on an LSB conforming system.

- `libchk`

  This test suite checks the libraries required by the LSB to ensure they contain the interfaces and symbol versions as specified by the LSB.

- `runtimetests`

  This test suite measures the behavior of the interfaces provided by the GNU/Linux system. This is the largest of the test suites, and is actually broken down into several components, which are referred to collectively as the runtime tests. These tests are derived from the test suites used by the Open Group for UNIX branding.

LSB Packaging

- `pkgchk`

  This test examines an RPM format package to ensure it conforms to the LSB.

- `pkginstchk`

  This test suite is used to ensure that the package management tool provided by a GNU/Linux system will correctly install LSB conforming packages. This suite is still in early stages of development.

LSB Application

- `appchk`

  This test performs a static analysis of an application to ensure that it only uses libraries and interfaces specified by the LSB.

- `dynchk`

  This test is used to measure an applications use of the LSB interfaces during its execution, and is the subject of this paper.

2 The database

The LSB Specification contains over 6600 interfaces, each of which is associated with a library and a header file, and may have parameters. Because of the size and complexity of the data describing these interfaces, a database is used to maintain this information.

It is impractical to try and keep the specification, test suites and development libraries and headers synchronized for this much data. Instead, portions of the specification and tests, and all of the development headers and libraries are generated from the database. This ensures that as changes are made to the database, the changes are propagated to the other parts of the project as well.

Some of the relevant data components in this DB are Libraries, Headers, Interfaces, and Types. There are also secondary components and relations between all of the components. A short description of some of these is needed before moving on to how the dynchk test is constructed.

2.1 Library

The LSB specifies 17 shared libraries, which contains the 6600 interfaces. The interfaces in each library are grouped into logical units called a LibGroup. The LibGroups help to organize the interfaces, which is very useful in the written specification, but isn't used much elsewhere.

2.2 Interface

An Interface represents a globally visible symbol, such as a function, or piece of data. Interfaces have a Type, which is either the type of the global data or the return type of the function. If the Interface is a function, then it will have zero or more Parameters, which form a

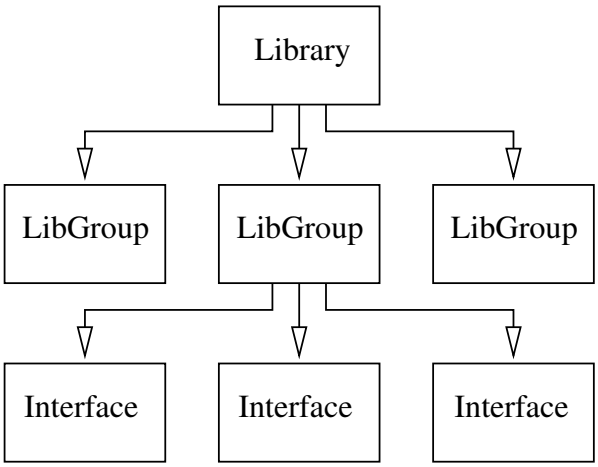

Figure 1: Relationship between Library, LibGroup and Interface

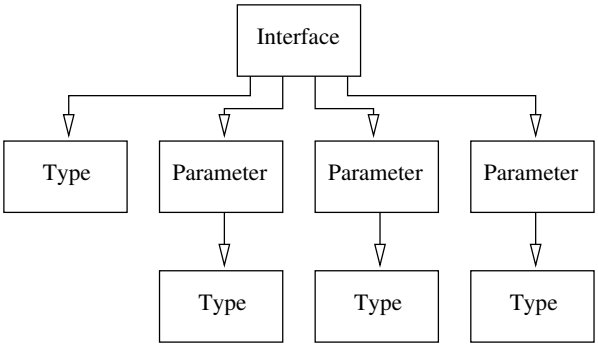

Figure 2: Relationship between Interface, Type and Parameter

set of Types ordered by their position in the parameter list.

2.3 Type

As mentioned above, the database contains enough information to be able to generate header files which are a part of the LSB development tools. This means that the database must be able to represent Clanguage types. The Type and TypeMember tables provide these. These tables are used recursively. If a Type is defined in terms of another type, then it will have a base type that points to that other type.

For structs and unions, the TypeMemeber table is used to hold the ordered list of members. Entries in the TypeMember table point back to the Type table to describe the type of each member. For enums, the TypeMember table is also used to hold the ordered list of values.

Tid	Ttype	Tname	Tbasetype
1	Intrinsic	int	0
2	Pointer	int *	1

Table 1: Example of recursion in Type table for int *

```
struct foo {
  int    a;
  int   *b;
}
```

Figure 3: Sample struct

Tid	Ttype	Tname	Tbasetype
1	Intrinsic	int	0
2	Pointer	int *	1
3	Struct	foo	0

Table 2: Contents of Type table

The structure shown in Figure 3 is represented by the entries in the Type table in Table 2 and the TypeMember table in Table 3.

2.4 Header

Headers, like Libraries, have their contents arranged into logical groupings known a HeaderGroups. Unlike Libraries, these HeaderGroups are ordered so that the proper sequence of definitions within a header file can be maintained. HeaderGroups contain Constant definitions (i.e. #define statements) and Type definitions. If you examine a few well designed header files, you will notice a pattern of a comment followed by related constant definitions and type definitions. The entire header file can be viewed as a repeating sequence of this pat-

Tmid	TMname	TMtypeid	TMposition	TMmemberof
10	a	1	0	3
11	b	2	1	3

Table 3: Contents of TypeMember

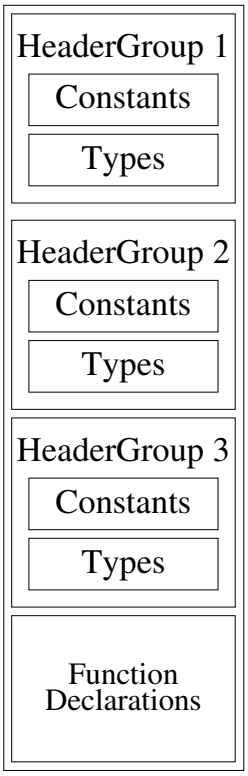

Figure 4: Organization of Headers

tern. This pattern is the basis for the HeaderGroup concept.

2.5 TypeType

One last construct in our database should be mentioned. While we are able to represent a syntactic description of interfaces and types in the database, this is not enough to automatically generate meaningful test cases. We need to add some semantic information that better describes how the types in structures and parameters are used. As an example, `struct sockaddr` contains a member, `sa_family`, of type unsigned short. The compiler will of course ensure that only values between 0 and $2^{16} - 1$ will be used, but only a few of those values have any meaning in this context. By adding the semantic information that this member holds a socket family value, the test generator can cause the value found in `sa_family` to be tested against the legal socket families values (`AF_INET`, `AF_INET6`, etc), instead of just ensuring the value falls between 0 and $2^{16} - 1$, which is really just a noop test.

Example TypeType entries

- `RWaddress`

  An address from the process space that must be both readable and writable.

- `Rdaddress`

  An address from the process space that must be at least readable.

- `filedescriptor`

  A small integer value greater than or equal to 0, and less than the maximum file descriptor for the process.

- `pathname`

  The name of a file or directory that should be compared against the Filesystem Hierarchy Standard.

2.6 Using this data

As mentioned above, the data in the database is used to generate different portions of the LSB project. This strategy was adopted to ensure

these different parts would always be in sync, without having to depend on human intervention.

The written specification contains tables of interfaces, and data definitions (constants and types). These are all generated from the database.

The LSB development environment[1] consists of stub libraries and header files that contain only the interfaces defined by the LSB. This development environment helps catch the use of non-LSB interfaces during the development or porting of an application instead of being surprised by later test results. Both the stub libraries and headers are produced by scripts pulling data from the database.

Some of the test suites described previously have components which are generated from the database. `Cmdchk` and `libchk` have lists of commands and interfaces respectively which are extracted from the database. The static application test tool, `appchk`, also has a list of interfaces that comes from the database. The dynamic application test tool, `dynchk`, has the majority of its code generated from information in the database.

3 The Dynamic Checker

The static application checker simply examines an executable file to determine if it is using interfaces beyond those allowed by the LSB. This is very useful to determine if an application has been built correctly. However, is unable to determine if the interfaces are used correctly when the application is executed. A different kind of test is required to be able to perform this level of checking. This new test must interact with the application while it is running, without interfering with the execution of the application.

This new test has two major components: a mechanism for hooking itself into an application, and a collection of functions to perform the tests for all of the interfaces. These components can mostly be developed independently of each other.

3.1 The Mechanism

The mechanism for interacting with the application must be transparent and noninterfering to the application. We considered the approach used by 3 different tools: abc, ltrace, and fakeroot.

- `abc`—This tool was the inspiration for our new dynamic checker. `abc` was developed as part of the SVR4 ABI test tools. `abc` works by modifying the target application. The application's executable is modified to load a different version of the shared libraries and to call a different version of each interface. This is accomplished by changing the strings in the symbol table and `DT_NEEDED` records. For example, `libc.so.1` is changed to `LiBc.So.1`, and `fread()` is changed to `FrEaD()`. The test set is then located in `/usr/lib/LiBc.So.1`, which in turns loads the original `/usr/lib/libc.so.1`. This mechanism works, but the requirement to modify the executable file is undesirable.
- `ltrace`—This tool is similar to `strace`, except that it traces calls into shared libraries instead of calls into the kernel. `ltrace` uses the ptrace interface to control the application's process. With this approach, the test sets are located in a separate program and are invoked by stopping the application upon

[1] See the May Issue of *Linux Journal* for more information on the LSB Development Environment.

entry to the interface being tested. This approach has two drawbacks: first, the code required to decode the process stack and extract the parameters is unique to each architecture, and second, the tests themselves are more complicated to write since the parameters have to be fetched from the application's process.

- `fakeroot`—This tool is used to create an environment where an unprivileged process appears to have root privileges. `fakeroot` uses `LD_PRELOAD` to load an additional shared library before any of the shared libraries specified by the `DT_NEEDED` records in the executable. This extra library contains a replacement function for each file manipulation function. The functions in this library will be selected by the dynamic linker instead of the normal functions found in the regular libraries. The test sets themselves will perform tests of the parameters, and then call the original version of the functions.

We chose to use the `LD_PRELOAD` mechanism because we felt it was the simplest to use. Based on this mechanism, a sample test case looks like Figure 5.

One problem that must be avoided when using this mechanism is recursion. If the above function just called `read()` at the end, it would end up calling itself again. Instead, the `RTLD_NEXT` flag passed to `dlsym()` tells the dynamic linker to look up the symbol on one of the libraries loaded after the current library. This will get the original version of the function.

3.2 Test set organization

The test set functions are organized into 3 layers. The top layer contains the functions that are test stubs for the LSB interfaces. These functions are implemented by calling the functions in layers 2 and 3. An example of a function in the first layer was given in Figure 5.

The second layer contains the functions that test data structures and types which are passed in as parameters. These functions are also implemented by calling the functions in layer 3 and other functions in layer 2. A function in the second layer looks like Figure 6.

The third layer contains functions that test the types which have been annotated with additional semantic information. These functions often have to perform nontrivial operations to test the assertion required for these supplemental types. Figure 7 is an example of a layer 3 function.

Presently, there are 3056 functions in layer 1 (tests for `libstdc++` are not yet being generated), 106 functions in layer 2, and just a few in layer 3. We estimate that the total number of functions in layer 3 upon completion of the test tool will be on the order of several dozen. The functions in the first two layers are automatically generated based on the information in the database. Functions in layer 3 are hand coded.

3.3 Automatic generation of the tests

In Table 4, is a summary of the size of the test tool so far. As work progresses, these numbers will only get larger. Most of the code in the test is very repetitive, and prone to errors when edited manually. The ability to automate the process of creating this code is highly desirable.

Let's take another look at the sample function from layer 1. This time, however, lets replace some of the code with a description of the information it represents. See Figure 8 for this parameterized version.

All of the occurrences of the string *read* are

```
  ssize_t read (int arg0, void *arg1, size_t arg2) {
    if (!funcptr)
        funcptr = dlsym(RTLD_NEXT, "read");
    validate_filedescriptor(arg0, "read");
    validate_RWaddress(arg1, "read");
    validate_size_t(arg2, "read");
    return funcptr(arg0, arg1, arg2);
  }
```

Figure 5: Test case for read() function

```
void validate_struct_sockaddr_in(struct sockaddr_in *input,
                                 char *name) {
  validate_socketfamily(input->sin_family,name);
  validate_socketport(input->sin_port,name);
  validate_IPv4Address((input->sin_addr), name);
}
```

Figure 6: Test case for validating `struct sockaddr_in`

Module	Files	Lines of Code
libc	752	19305
libdl	5	125
libgcc_s	13	262
libGL	450	11046
libICE	49	1135
libm	281	6568
libncurses	266	6609
libpam	13	335
libpthread	82	2060
libSM	37	865
libX11	668	16112
libXext	113	2673
libXt	288	7213
libz	39	973
structs	106	1581

Table 4: Summary of generated code

actually just the function name, and could have been replaced also.

The same thing can be done for the sample function from layer 2 as is seen in Figure 9.

These two examples, now represent templates that can be used to create the functions for layers 1 and 2. From the previous description of the database, you can see that there is enough information available to be able to instantiate these templates for each interfaces, and structure used by the LSB.

The automation is implemented by 2 perl scripts: `gen_lib.pl` and `gen_tests.pl`. These scripts generate the code for layers 1 and 2 respectively.

Overall, these scripts work well, but we have run into a few interesting situations along the way.

3.4 Handling the exceptions

So far, we have come up with an overall architecture for the test tool, selected a mechanism that allows us to hook the tests into the running application, discovered the pattern in the test functions so that we could create a template for

```
void validate_filedescriptor(const int fd, const char *name) {
   if (fd >= lsb_sysconf(_SC_OPEN_MAX))
       ERROR("fd too big");
   else if (fd < 0)
       ERROR("fd negative");
   }
```

Figure 7: Test case for validating a filedescriptor

```
 return-type read (list of parameters) {
     if (!funcptr)
         funcptr = dlsym(RTLD_NEXT, "read");
     validate_parameter1 type(arg0, "read");
     validate_parameter2 type(arg1, "read");
     validate_parameter3 type(arg2, "read");
     return funcptr(arg0, arg1, arg2);
 }
```

Figure 8: Parameterized test case for a function

automatically generating the code, and implemented the scripts to generate all of the tests cases. The only problem is that now we run into the real world, where things don't always follow the rules.

Here are a few of the interesting situations we have encountered

- `Variadic Functions`

  Of the 725 functions in libc, 25 of them take a variable number of parameters. This causes problems in the generation of the code for the test case, but most importantly it affects our ability to know how to process the arguments. These function have to be written by hand to handle the special needs of these functions. For the functions in the `exec`, `printf` and `scanf` families, the test cases can be implemented by calling the varargs form of the function (`execl()` can be implemented using `execv()`).

- `open()`

  In addition to the problems of being a variadic function, the third parameter to `open()` and `open64()` is only valid if the `O_CREAT` flag is set in the second parameter to these functions. This simple exception requires a small amount of manual intervention, so these function have to be maintained by hand.

- `memory allocation`

  One of the recursion problems we ran into is that memory will be allocated within the `dlsym()` function call, so the implementation of one test case ends up invoking the test case for one of the memory allocation routines, which by default would call `dlsym()`, creating the recursion. This cycle had to be broken by having the test cases for these routines call libc private interfaces to memory allocation.

- `changing memory map`

```
void validate_struct_structure name(struct structure name
   *input, char *name) {
     validate_type of member 1(input->name of member 1, name);
     validate_type of member 2(input->name of member 2, name);
     validate_type of member 3((input->name of member 3), name);
}
```

Figure 9: Parameterized test case for a struct

Pointers are validated by making sure they contain an address that is valid for the process. `/proc/self/maps` is read to obtain the memory map of the current process. These results are cached, for performance reasons, but usually, the memory map of the process will change over time. Both the stack and the heap will grow, resulting in valid pointers being checked against a cached copy of the memory map. In the event a pointer is found to be invalid, the memory map is re-read, and the pointer checked again. The `mmap()` and `munmap()` test cases are also maintained by hand so that they can also cause the memory map to be re-read.

- `hidden ioctl()s`

  By design, the LSB specifies interfaces at the highest possible level. One example of this, is the use of the termio functions, instead of specifying the underlying `ioctl()` interface. It turns out that this tool catches the underlying `ioctl()` calls anyway, and flags it as an error. The solution is for the termio functions the set a flag indicating that the `ioctl()` test case should skip its tests.

- `Optionally NULL parameters`

  Many interfaces have parameters which may be NULL. This triggerred lots of warnings for many programs. The solution was to add a flag that indicated that the Parameter may be NULL, and to not try to validate the pointer, or the data being pointed to.

No doubt, there will be more interesting situations to have to deal with before this tool is completed.

4 Results

As of the deadline for this paper, results are preliminary, but encouraging. The tool is initially being tested against simple commands such as ls and vi, and some X Windows clients such as xclock and xterm. The tool is correctly inserting itself into the application under test, and we are getting some interesting results that will be examined more closely.

One example is vi passes a NULL to `__strtol_internal` several times during startup.

The tool was designed to work across all architectures. At present, it has been built and tested on only the IA32 and IA64 architectures. No significant problems are anticipate on other architectures.

Additional results and experience will be presented at the conference.

5 Future Work

There is still much work to be done. Some of the outstanding tasks are highlighted here.

- Additional `TypeTypes`

  Semantic information needs to be added for additional parameters and structures. The additional layer 3 tests that correspond to this information must also be implemented.

- Architecture-specific interfaces

  As we found in the LSB, there are some interfaces, and types that are unique to one or more architectures. These need to be handled properly so they are not part of the tests when built on an architecture for which they don't apply.

- Unions

  Although Unions are represented in the database in the same way as structures, the database does not contain enough information to describe how to interpret or test the contents of a union. Test cases that involve unions may have to be written by hand.

- Additional libraries

  The information in the database for the graphics libraries and for `libstdc++` is incomplete, therefore, it is not possible to generate all of the test cases for those libraries. Once the data is complete, the test cases will also be complete.

Proceedings of the Linux Symposium

Volume One

July 21st–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*