Linux Distributions and Applications Analysis During Linux Standard Base Development

Denis Silakov
Institute for System Programming
at the Russian Academy of Sciences
Moscow, Russian Federation
Email: silakov@ispras.ru

***Abstract*—This paper presents a novel approach of collecting and managing information about existing Linux distributions and applications necessary for standardizing Linux ecosystem during Linux Standard Base specification development. The approach is based on the usage of database designed on the basis of abstract model of the Linux ecosystem. The process of database schema design is described as well as design of tools for filling it with data and analytical instruments.**

***Index Terms*—Data management, Reverse engineering, Software requirements and specifications, Software standards.**

I. Introduction

One of the main aims of software standards is to unify different implementations which solve the same problems and perform the same operations. Standards on programming languages try to describe a set of compiler-independent language features, SQL standards – DBMS-independent SQL features, specifications of operating systems try to describe common interfaces of different system distributions (either API, like Single Unix Specification [1], or ABI, like LSB [2]).

Many standards suggest new features and solutions which are not implemented yet in real systems; quite often such suggestions are based on some theoretical research results. However, a problem can arise if systems exist that already propose their own approaches to the same issues. In this case it's a common practice for standard developers to take implementation which is known to be de-facto standard. But if there is no such implementation (i.e. there are several competitive approaches) then standardization becomes rather difficult. It's very likely that the standard developed will not be accepted by people if most systems are not going to support it (providing alternative implementations instead). In such situation standardization attempt can lead to a very restricted specification which defines only those features that are accepted by all competitors (like Single Unix Specification, discussed below) or can even fail (like Xbase standardization efforts [3]). Thus, if there are several competitive systems, standard developers should analyze them all and try to construct a set of features which will satisfy all parties – both target systems developers and users.

The problem of existence of many competitive systems is very relevant in the Open Source world. In the 90th, there existed dozen UNIX systems which sometimes significantly differed from each other. This fact led to so called 'UNIX Wars' and didn't give to UNIX any advantages, indeed. Single UNIX Specification (SUS) was created as an attempt to unify different UNIX systems. During SUS development not only operating systems were analyzed, but also applications needs were investigated. A research of 50 applications was performed in the 1993 and 130 interfaces were discovered widely used by programs but not described in any specification [4]. However, while being rather successfull in unifying system software, SUS didn't contain enough interfaces to allow developers to create portable applications [5]. The thing is that the standard doesn't concern many important areas, the main of which is graphical user interface. Addressing the numbers, modern systems provide hundreds of thousands or even millions of interfaces that can be used by applications, and only less than 2000 are covered by the latest SUS version.

Nowdays the problem of many different competitive systems has appeared in the Linux world. There are hundreds of different Linux distributions, which are sometimes also significantly different. In order to prevent Linux world fragmentation, The Linux Foundation [6] (a nonprofit consortium dedicated to fostering the growth of Linux) develops a standard called Linux Standard Base (LSB) which aims to describe elements that can be found in any LSB-compatible distribution. Unlike SUS, LSB is a binary level standard, and its main part specifies Application Binary Interface (ABI). This should help developers to create portable applications that can be executed on any LSB-compatible system without recompilation.

LSB concerns not only core system functions, but all interfaces that are claimed by application developers and that are mature enough to be included in the specification. Thus, there are two main tasks here – the first is to determine which elements are 'mature enough' and the second is to select interfaces that are mostly important for applications. The key problem here lies in the size of data to be analyzed – there are hundreds of distributions, thousands of applications and millions of binary interfaces. LSB now describes about 40.000 binary interfaces, and this is still not enough for most applications. Taking into account SUS experience, one can notice that manual analysis and selection of thousands of interfaces to be included in the specification may require dozens man-years. Therefore it is necessary to have an approach of collecting and managing data about Linux applications and distributions that will allow to standard developers to easily get information

about implementation of different interfaces in real systems, their usage by applications, and, maybe, some additional useful details (e.g. existing documentation).

The remainder of the paper is structured as follows: Section 2 describes the principles of the Linux Standard Base development. Section 3 introduces database design based on Linux ecosystem abstract model. Section 4 describes the fundamentals of binary files analysis in Linux on which the tools are based that fill the database with data. Section 5 adds some words about analysis of programs that use interpreted languages. Section 6 introduces analytical tools used to make decisions on standardization. Finally, Section 7 summarizes the ideas.

II. Linux Standard Base

The main purpose of the Linux Standard Base specification is to make applications development easier for vendors. Differences between Linux distributions lead to the fact that almost every application should be recompiled for every particular system, since one cannot guarantee binary compatibility of different distributions. This is not very problematic for open source applications – usually developers of every distribution that provide binary packages simply compile all necessary applications by themselves. Even developers of source-based distributions before including applications to the stable version of the system perform different tests to unsure that these applications can be compiled and successfully launched inside the distribution environment. The situation is more difficult for proprietary applications – in this case vendors cannot simply give the source code to every distribution vendor, so they have to create binary packages for every distributions they are interested in. To be sure, the number of supported distributions for such applications is usually very small.

However, all distributions contain a lot of similar components, so they definitely have a lot of common things. LSB tries to collect and describe elements (libraries, functions, commands, etc.) that are the same in most distributions. It is impossible to cover all existing distributions, indeed; there are a lot of systems created for particular purposes which really have quite a few intersections. But most popular distributions (including those that have significant market share – a rather important fact for proprietary applications' vendors) do contain lots of similar elements. LSB focuses on binary symbols and libraries, also defining some commands and shell utilities that might be useful for application installation. Thus, the main idea of LSB is that if some application uses only elements specified by the standard (i.e. it is LSB compatible application) then it can be installed and launched from the same binary files on any distribution that provides all such elements (i.e. it is LSB compatible distribution). Surely, this works only inside the same hardware architecture, since binary file formats differ on different architectures (of course, one can launch, for example, files targeted for x86 platform on x86-64 systems, but only if the latter contain all necessary 32bit libraries in addition to their 64bit variants).

Formally, the Linux Foundation has two main criteria which should be met for every binary interface going to be included in the LSB specification:

- The interface should be stable – i.e. its developers can guarantee that the same interface in different library versions has the same behavior (and will remain the same in future versions). Note that for cases when modification of interface behavior is required, binary symbols versions can be used.
- The interface should be present in the main modern distributions. Linux Foundation doesn't aim to cover all existing distributions, but focuses on those who have the major market share – SUSE, Red Hat, Oracle Enterprise Linux (based on Red Hat), Ubuntu, Debian, Mandriva and some others.
- The interface should be requested by applications. It's a fact, indeed, that even some interfaces having been provided by distributions for a long time, are not actually used in real-life applications.

There are also some additional criteria (not obligatory, but very desirable), concerning existing documentation, tests, etc. Though they are rather important, their analysis is out of scope of this paper and we'll not consider them here.

For modules of interpreted languages requirements are the same.

The final decision is done by human after consulting with developers of components going to be included in the specification. It is the first criterion that should be guaranteed by developers – nobody else can predict the future. And even if all criteria mentioned above are met, the function can be declined by some other reasons - for example, as being unstable, unsafe from security point of view, etc. To say the truth, even if developers are not going to change the interface behavior, this can be simply done by distribution vendors then building the component for their systems – maybe not directly, but as a side effect of some other patches. But such situations are determined by LSB certification tests and vendor should change the behavior back to the specified one if he wants to be LSB compatible.

However, before consulting the actual developers of components that are candidates to be included in the specification, LSB analysts should prepare concrete suggestions about interfaces they'd like to include. This actually means that the second and the third criteria (i.e. presence in distributions and usage by applications) should be investigated. This is not the thing which component developers supposed to analyze (though it may be rather interesting for them, too). On the basis of such investgations, first, a set of libraries is determined which are mostly used by applications and are present in every distribution. Then a list of binary symbols inside each library is created which meet the same criteria. Analysts can also perform some additional tasks here – for example, check interface documentation and tests, if any. Only after that a list of symbols is presented to component developers for comments. It is usually not difficult for them to check the list and to say which interfaces can be safely included, which

are not (and why) and maybe give some additional comments (for example, suggest more interfaces to be included as they usually accompany those that are already in the list).

Thus, before component developers are involved, LSB developers should analyze dozens of distributions, hundreds of applications and millions of interface and to prepare a list of symbols that should be considered as LSB candidates. As was mentioned in the Introduction, the key problem here lies in the amount of data – only manual selection of dozens of thousands interfaces to be considered as LSB candidates may take dozens man years. This is rather expensive and doesn't look to be reasonable, especially keeping in mind the fact that Linux is evolving and new libraries and interfaces constantly appear. Thus, LSB developers need a way to quickly collect, easily manage and efficiently analyze large amount of data on Linux distributions and applications.

III. Abstract Model of the Linux Ecosystem

An approach of collecting and managing large sets of distributions and applications data was developed in the ISP RAS under the contract with the Linux Foundation. The approach suggests development of the infrastructure consisting of a database storing necessary data, tools used to collect information and upload it to the database and analytical tools representing collected data in the form convenient for analysts.

The starting point of the infrastructure development is the database schema design. To develop a schema, one should sharply define what exactly will be subjected to analysis. It is necessary to define entities and their properties which will be required during analysis, define relations between these entities and represent it in a way suitable to database setup.

Appropriate database schema for the LSB purposes was designed on the basis of analysis of LSB needs and criteria described above as well as distributions and applications structure. On the basis of this analysis, the abstract model of the Linux ecosystem was created. Entity-Relationship diagram of the model is presented on Fig.1. The main entities here are distributions and applications. Distributions are divided to components (corresponding to real upstream components from which every distribution is constructed, such as glibc, X11, GTK, Qt, etc.). Each component may contain libraries, commands and modules of interpreted languages. Libraries, in turn, provides interfaces which are grouped by classes (for C++ interfaces). As for applications, every program can require libraries directly (by their sonames), binary interfaces and interpreted languages modules.

One can notice that some relations (for example, between distributions and components) are marked as 'one-to-many' though it could reasonable to mark them as 'many-to-many' instead (especially taking into account the fact mentioned by us before – the key feature of distributions on which LSB is based is that different systems actually contain the same components). Nevertheless, from the our model point of view the only thing which remains the same for the same component in different distributions is its name. All other properties (version, packages which form the component, etc.) are different. In

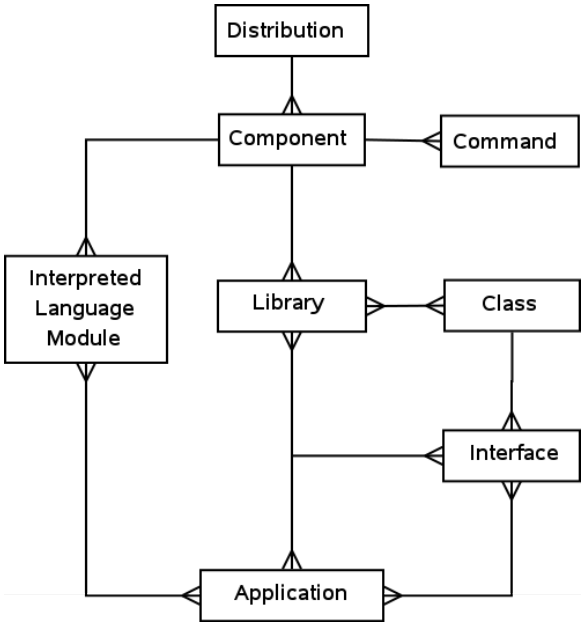

Fig. 1. Distributions and Applications Entity-Relationship Diagram

such situations the decision should be made on the basis of actual technology on which the database will be based. In case of LSB, the relational database was supposed to be used, so this question transforms to the problem of normalization; though normalization makes the data managing easier in some aspects, it usually decrease the speed of achieving necessary data. In our case very complicated requests are sometimes required, but all operations perfromed by analysts consist of selection of different information. The data was not supposed to be added changed manually, so we decided to denormalize the schema in some areas to increase the speed of requests (and, sequently, we had to create more complicated tools that upload and manage the data). Thus, we consider components from different distributions as different components, not regarding their names, versions, etc. Groupings by names, versions and other parameters are perfromed by analytical tools if it is needed for report creation.

As for elements properties, there are quite a few of them that are required for analysis. These are binary symbols versions, libraries sonames (discussed in the section below) and commands locations. There are some additional properties useful for analysis (such as applications and distributions versions) that also should be stored in the database.

Database alone is of no use, indeed, if it is not filled with data. ISP RAS team has developed a set of tools that collect data about Linux distributions and applications and upload it to the database [7]. There are actually two sets of tools: one is for data collection and the other is for data upload. Tools from the first set analyses shared objects and executables of applications and distributions. These tools determines binary symbols and libraries provided by distributions and used by applications. The analysis algorithm is actually the same for distributions and applications. However, for distributions it

is important which interfaces and sonames are provided by its libraries. For applications, we should know which binary symbols should be present in the system to execute its binary files. We should also keep in mind that applications can ship some libraries with themselves; if some symbol is required by some executable file and provided by a library shipped with the application, then this symbol is not considered as external dependency. Such analysis of collected information (including analysis of actual external dependencies of the application) is performed by tools uploading information to the database.

Next section presents the fundamentals of binary executable files and shared objects analysis in Linux on which the tools are based that collect applications and distributions data. In addition. Section 5 concerns analysis of programs that use interpreted languages as well as analysis of interpreters with their environments implemented in particular distributions.

IV. Binary Files Analysis

A. Executable and Linking Format

Almost all shared objects and binary executables in Linux have ELF (Executable and Linking Format) file format (there is one more format for UNIX-like systems, COFF – Common Object File Format, but it is not used widely in Linux at the moment). The ELF format for UNIX System V is described in the System V Application Binary Interface specification [8]. Some additional aspects for Linux are specified in the Linux Standard Base Core Specification [9]. Content of the ELF files can be explored using functions from the libdl library, which provides C API to access internal structure of such files. There also standard utilities, such as readelf, nm or objdump, that can present ELF file contents in a human-readable form.

Every ELF file has a header containing information that can be useful during file analysis. At least the following attributes are usually important:

- Class – ELF32 for 32-bit platforms (and for IBM S/390 formally known as 31-bit platform), ELF64 for 64-bit ones.
- OS/ABI – 13 different OS/ABI types are defined at the moment, including 'Linux' and 'UNIX System V ABI' (the latter is the default type supported by all systems including Linux).
- Type – Executable, shared object, relocatable or core file.
- Machine – Actual architecture for which this file is intended for. There are more than 70 possible values for this attribute (and x86 is only one of them!). Note that in some cases this field itself is not enough to define target acrchitecture precisely. For example, both IBM S/390 and IBM S/390x files have 'IBM S/390' value in this field. In such cases 'Class' field should be used in addition to 'Machine' – in our example S/390 files belong to the ELF32 class, and IBM S/390x files - to the ELF64 one.

After the header, different ELF file sections and tables follow. LSB describes about 40 sections of 17 different types that may be present in ELF file. We'll shortly describe here those that are necessary for detection of binary symbols exported by shared libraries as well as symbols and libraries required by application to be executed.

B. ELF Symbol Tables

In order to find out which external interfaces are required by executable file or which symbols are exported by shared object, one should explore symbol tables of the file which contain lists of used binary symbols. Symbol tables are stored in either '.dynsym' or '.symtab' section of the file. The following properties can be obtained for every symbol from the symbol tables:

- Type – for dependencies analysis, 'function' and 'object' types are of interest ('object' type designates data interface, such as global variable or alias).
- Binding – local, weak, global or some processor-specific or operating system-specific values.
- Visibility – default, internal, hidden or protected.
- Index – ABS (absolute), UND (undefined) or some numeric value.
- Offset value – the offset has non-zero value if the symbol is implemented inside the file and zero value otherwise.

We will not consider processor or system-specific bindings here. Symbols with local binding in shared objects are visible only inside the component where they are defined. As for executables, local binding of a symbols means that the appropriate interface is implemented inside the file.

Global and weak symbols in shared objects are symbols that are visible externally. The difference between 'global' and 'weak' binding lies in restricions on symbol names – when several relocatable object files are combined by link editor, there must be no definitions of 'global' symbols with the same name, while multiple definitions of 'weak' interfaces are allowed (as well as precense of one 'global' and several 'weak' symbols with the same name). That's why applications using weak symbols are considered to be unreliable – changes in the runtime environment might cause their execution to fail; weak symbols are intended primarily for use in system software.

In case of executable files, global symbols are actually external references that should be resolved by link editor before the execution starts.

The actual visibility of symbols with 'default' visibility value is as specified by the symbol's binding type. That is, global and weak symbols in shared objects are visible outside the file, local symbols are not. Symbols with 'protected' visibility can be also accessed from the outside, but unlike default symbols with global or weak binding they are not preemptable. Hidden and internal symbols can be accessed only within the file where they are defined. As for executables, their external references have 'default' visibility.

C. ELF Dynamic Sections

According to LSB, if an object file participates in dynamic linking, its program header table shall have an element of a special type, PT_DYNAMIC. Segment represented by this element contains so called 'dynamic' section, which, in turn, contains different entries necessary for linker. Some of them

are auxiliary, the others represent the internal structure of the file, but there is one kind of such entries which is vital for its interaction with the system environment. We mean DT_NEEDED entries containing library names that should be present in the system in order to successfully link the current file. More precisely, DT_NEEDED entries reference libraries' 'sonames'.

As denoted at LinuxQuestions.org [10], the point of the soname is for the creator of the library to be able to provide version compatibility information to the system. I.e. if a new version of the library is released which is fully compatible with the old one, developers can assign the same soname to it; this will make two versions of the library identical from the linker point of view. Thus, library can be safely updated without touching the programs that use it (surely, if developers haven't broken anything). Soname of every library is stored in a special dynamic entry of its ELF file, called DT_SONAME.

Thus, every ELF file may have a list of DT_NEEDED dependencies, telling the linker which libraries should be loaded to resolve symbols dependencies of this file discussed in the section above. The process is recursive – all dependencies of the referenced libraries should also be resolved, and so on. If the linker cannot satisfy any dependency, the linking process will be stopped by error; a list of sonames known to the system linker can be obtained by calling 'ldconfig' with '-p' option. If a binary symbol needed by the file is not found among libraries collected, the linking will fail, too.

However, a set of DT_NEEDED entries should not be considered as a reliable list of libraries which are really required by application. There are two common problems that can be found in real-life applications:

- DT_NEEDED list may contain library whose symbols are not really used by the file.
- DT_NEEDED list may not contain library whose symbols are used by the file.

The first problem is caused by the fact that default link editor behavior is to simply put all libraries pushed to it through '-l' option to the DT_NEEDED list. But such a list taken from the command line often contains superfluous names. Many developers do not want to form such list manually and use some auxiliary tools, such as pkg-config [11]. However, simplicity here sometimes leads to extra dependencies. A common example is about compiling programs using GTK libraries – with pkg-config developers can simply get necessary linker options by calling 'pkg-config --libs gtk+-2.0'. The tool will fairly return the line containing all libraries concerning GTK+-2.0. The list contains about 10 libraries, and very often only some of them are really used by application.

One can object that libraries in the list constructed by pkg-config are usually closely interconnected, so during dynamic linking they will be loaded anyway. This is true, but only for a certain system where the application is built. No one can guarantee that on the other system the interconnection between libraries is the same. The worst thing here is that some of the libraries not actually required by application but included in its DT_NEEDED list can be missing on other systems. In this case the application will fail to execute on such systems, even though they do have all really necessary libraries and symbols.

The problem with unnecessary dependencies is known to many Linux developers, including those who is responsible for link editor development. The 'ld' linker has '--as-needed' option which forces it to detect libraries really required by files and to construct DT_NEEDED list from such libraries only. However, this feature is still rather unstable and known to cause problems with many applications [12]. A list of DT_NEEDED entries which are unnecessary from the linker point of view can be obtained by using 'ldd' command with '-u' option. There is also a 'checklib' tool [13], used by some distribution developers to check their packages.

The second problem is opposite to the first one and usually appears when developers manually form list of libraries that should be placed in the DT_NEEDED entries. In this case it is possible to omit some libraries in the list expecting that they will be loaded as dependencies of other libraries referenced in the list. However, applications compiled with such tricks can also fail to execute on systems where interconnections between libraries differ.

V. Scripts Analysis

Creating binary executable files is not the only form of distributing applications. Many developers use interpreted languages (such as Perl, Python or Ruby) to create at least some parts of their applications. In order to execute such application on some distribution, the latter should provide appropriate language interpreter and, maybe, some additional staff (libaries, modules, etc.). LSB takes this fact into account and specifies some aspects of Perl and Python interpreters and environment that should be provided by LSB-compliant distribution. More particular, LSB specifies minimal versions of Perl and Python interpreters that should be provided by distributions and sets of modules that should be present in the system.

Thus, for analytical purposes it is important to know which modules are provided by distributions and which are used by applications. As for python, it has a separate module called 'modulefinder' that allows to get the list of modules the application depends on. However, this module calculates not only direct dependencies (i.e. it recursively calculates dependencies of all modules required by application). This is actually not exact what is required for analysis; so for LSB puproses a modified version of the module called lsb-modulefinder is developed which reports direct dependencies only. As for Perl language modules, perldeps.pl script from the rpm-build package can is used.

Note, however, that some interpreted languages provide constructions that significantly complicate program analysis. For example, in Python the exported name of the module can be constructed in runtime, as well as a name of imported module; moreover, module imports can be enclosed with different 'try' or 'if' statements, so it is possible that the same application will use different modules in different system environments. Surely, some similar tricks are available in binary

executables, too. Among possible examples we can mention the 'system' function that can execute any system command, 'syscall' function performing system calls and functions from the 'dlopen' library that allow to call specific interfaces from libraries hiding this fact from the dynamic loader. However, in case of binaries files we still can detect the usage of 'system' , 'syscall' or 'dlopen' symbols and the fact of their usage itself indicates that there can be some problems with application portability. In case of interpreted languages it is sometimes impossible to detect such tricky situations, so tools performing analysis of programs that use interpreted languages are still consider as experimental. Sometimes results obtained by these tools look unrealistic and require manual analysis of application source code. Fortunately, manual analysis of scripts is much more simple that the one for binary files, so analysts usually don't spend much time on it.

VI. Analysis Tools

Thus, we've designed a database and fill it with all necessary data. But though a database provides us with everything we need for analysis, it is not convenient to perform complex queries to the database every time we want to obtain some information. The tools are required which will take care of routine work to allow analysts to quickly obtain necessary information. Surely, one cannot develop a program which will satisfy all imaginable requests by terms of several mouse clicks. Thus, it is necessary to determine the typical kinds of reports that are used most often.

In case of Linux Standard Base, there are two typical requests that are very important for analytics:

- Which distributions provide the symbols given.
- Which applications use the symbols given.

Detailed elaboration of these generic requests are usually required to observe the data form different perspectives. For example, it is useful to know do all last versions of distributions provide the interface, does the symbols given are used only by games or which symbols are mostly important for proprietary applications.

The main program used for analysis during LSB development is LSB Navigator [14]. This is a web application which uses information from the database. Online working version can be accessed at the Linux Foundation portal. LSB ATK Manager – a standalone tool (i.e. it doesn't use the database) for testing, certifying and analyzing applications – also can create some analytical reports for applications. Both programs are distributed under the GPL version 2 license and can be freely downloaded from the Linux Foundation Bazaar repository [15].

A. LSB Navigator

Availabilities of representing different reports for any given sets of symbols is implemented in the LSB Navigator's 'Decision Making Support' section [16]. LSB Navigator also provides sections containing different statistical data, among which we'd like to mention the following:

- Top interfaces and libraries not included in LSB but used by applications.
- Applications 'LSB compatibility' – a rating of applications with respect to number of non-LSB elements used by them.

For the latter purpose LSB Navigator uses LSB specification database [17] that stores information about elements included in the specification.

LSB Navigator provides a set of filters for every statistical report, which can be used to observe the report from different perspectives, as it was mentioned above. More particular, the following filters are available:

- Application size – small, medium or large. This size is based on the number of external symbols required by application. Programs that use less than 100 symbols are considered as small, from 100 to 999 symbols – as medium, and products that require 1000 and more external interfaces are called 'large'. Note that size calculated by this way may not correspond to the actual size of application's binary files or source code. Applications with lots of lines in their source code may implement all necessary features inside themselves and decrease the interaction with system environment to some generic system calls.
- Application vendor. For proprietary applications, the actual vendor is stored. As for open source applications, the situation is not so straightforward. The thing is that the same application can have different external dependencies if compiled and built in different system environments, with different compiler options, etc. Therefore for opens source applications analysis we usually take the builds from particular distributions and mention those distribution vendor as application vendor.
- Functional category. Applications are divided on the following categories at the moment:
  - Accessibility and internationalization
  - Antivirus and Security
  - Emulators
  - Office and Desktop
  - Data Management
  - Development
  - Games
  - Multimedia and Graphics
  - Network
  - Science and Education
  - System Tools
  - X11 Utilities

  This set of values was developed on the basis of analysis of the same categorizations perfromed by different distributions (more particular, Gentoo and Debian were taken into account) and by large application vendors (e.g. Chiphopper program of the IBM [18]). Detailed description of these categorization is available on the LSB Navigator Help page.
- Licensing type – proprietary, open source or mixed (the

latter is for those vendors who have both proprietary and open licenses the same product).
- User interfaces – either GUI or non-GUI (command line interface only). Applications that provide both graphical and command line interfaces are considered as GUI ones.

LSB Navigators allow to select either single or multiple values for each filter, so it is possible to perform such requests as 'non-LSB interfaces mostly used by proprietary games, emulators and antiviruses' or 'large or small applications of VendorA, VendorB and VendorC vendors that are at the top of LSB compatibility rating'.

In addition to filters described above, LSB Navigator has Architecture filter. LSB is a binary level standard and on different architectures the sets of symbols may differ (for example, some symbols are available on 32bit platforms only, the others are specific for IBM S390 architecture, etc.). This filter allows to view reports for every architecture supported by LSB (7 architectures are supported at the moment – x86, x86-64, IA64, PPC32, PPC64, S390 and S390X).

As it was mentioned above, LSB Navigator uses the specification database to determine which elements are included in the standard. This specification database has special fields that support historical requests – i.e. it is possible to know which elements were included in a particular LSB version. Using this information, LSB Navigator allows to view reports from the different LSB versions retrospectives. This allows to observe, for example, how the number of LSB compatible applications changes in time, to decide, if the standard moves in the right direction.

B. LSB ATK Manager

Information about external dependencies of applications is important not only for standard developers, but also for application vendors. Analysis of their software not only can help to create LSB-compatible programs, but also helps to detect some particular problems in binary files discussed in the 'ELF Dynamic Sections' section, which is rather useful for all applications.

The data for any application can be gathered by tools described earlier, uploaded to the main LSB database and explored using LSB Navigator. But for LSB developers it doesn't make sense to have information about many different versions of the same application. It's very likely that these different versions don't differ vastly, indeed. It is possible to set up the database and LSB Navigator locally, but this is also not very convenient for single software vendor, since the setup procedure takes significant amount of time. That's why the possibility of application analysis was integrated in the LSB ATK Manager [19] – a tool that among other purposes has the one to help develop LSB-compatible applications.

Using ATK Manager, it is possible to analyse any given application (using its rpm or deb package or specifying application files directly) and to get the report similar to those produced by LSB Navigator for single application. ATK Manager doesn't require database; necessary data from the LSB specification database is shipped with the program, and data necessary for assigning interfaces to libraries is collected in runtime, using information from the system dynamic loader. Though the latter is not very fast and sometimes the analysis may take several minutes; in future, we plan to ship some additional information with ATK Manager which will allow to decrease the number of requests to system dynamic loader.

VII. Conclusion

This paper has suggested and approach of collecting and managing information about existing Linux distributions and applications necessary for analysis of applications portability among different Linux distributions. The method concerns all aspects that should be taken into account when exploring portability of binary executable files of the application written in C. The same aspects are also should be considered for applications written in C++; however, there is one C++ specific thing that is not covered yet – the structure of class virtual tables (though there are many C++ applications for which this is not important). Interpreted languages modules data and analysis tools are so mature yet and require more manual work, so this is one of the directions for development in future.

The approach suggests to store collected information in the database, but this is not the only possible solution, indeed. We've suggested a set of Linux ecosystem elements with their properties that should be placed in some data storage. We use a relational database as a such storage (with MySQL as DBMS) and we are satisfied by its performance and scalability. Though it should be mentioned that one of the reasons for such decision was the existence of stable and rather mature open source DBMS with convenient interface for different programming languages (in order to create tools to work with the database).

The approach developed was initially targeted to solve the problem of Linux ecosystem analysis during Linux Standard Base specification development. However, such analysis is very useful for application vendors themselves. Using the tools developed by Linux Foundation and ISP RAS (freely distributed under the GPLv2 license) they can analyze their applications portability. This possibility is used by proprietary vendors which don't want to place data about their applications to the central LSB database. As for analytical tools, though they initially aimed to answer the question 'can these symbols be added to the LSB' they can also answer another question very useful for software vendors – 'on which distributions the given binary build of the application can be launched'.

As LSB development process itself, the approach already proved to be very useful. The database at the moment contains data about 33 distributions and almost 1000 applications, which is quite enough to make decisions about binary symbols inclusion. Indeed, this data is the first thing that is checked by LSB analysts now when deciding whether to include given set of symbols to the specification or not; nothing is included without such checks.

References

[1] The Single UNIX Specification, Version 3. http://www.unix.org/version3/

[2] Linux Standard Base. https://www.linux-foundation.org/en/LSB
[3] Xbase – The Standards Effort. Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/XBase#The_standards_effort
[4] Andrew Josey. API Standards for Open Systems. http://www.opengroup.org/austin/papers/wp-apis.txt.
[5] Jerry Cashin. Bloom Fading From Posix Rose As Open Focus Shifts. Software Magazine, Volume 14, #3, March 1994. (http://findarticles.com/p/articles/mi_m0SMG/is_n3_v14/ai_15061435/)
[6] The Linux Foundation. https://www.linux-foundation.org
[7] Application and Distribution Data in the LSB Database. http://ispras.linux-foundation.org/index.php/Application_and_Distribution_Data
[8] System V Application Binary Interface – DRAFT – 17 December 2003. http://www.caldera.com/developers/gabi/2003-12-17/contents.html
[9] Linux Standard Base Core Specification 3.2. Executable and Linking Format. http://refspecs.linux-foundation.org/LSB_3.2.0/LSB-Core-generic/LSB-Core-generic.html#ELF-GENERIC
[10] LinuxQuestions.org – Library-related Commands and Files. http://wiki.linuxquestions.org/wiki/Library-related_Commands_and_Files#soname
[11] pkg-config Wiki. http://pkg-config.freedesktop.org/wiki/
[12] Gentoo Quality Assurance – '--as-needed' Introduction and Fixing Guide. http://www.gentoo.org/proj/en/qa/asneeded.xml
[13] Checklib Tool. http://swirl.redtux.org/checklib
[14] LSB Navigator. http://www.linux-foundation.org/navigator
[15] Linux Foundation Bazaar Repository. http://bzr.linux-foundation.org
[16] Decision Making Support in the LSB Navigator. http://ispras.linux-foundation.org/index.php/Administration_Mode_Howto#Decision_Making_Support
[17] LSB Specification Database. http://ispras.linux-foundation.org/index.php/LSB_Database_Home
[18] IBM Systems Application Advantage for Linux (Chiphopper). http://www.ibm.com/isv/go/chiphopper
[19] LSB Application Testkit Manager. http://ispras.linux-foundation.org/index.php/About_LSB_Application_Testkit_Manager