Improving Portability of Linux Applications by Early Detection of Interoperability Issues

Denis Silakov and Andrey Smachev

Institute for System Programming at the
Russian Academy of Sciences, Moscow, Russia
{silakov,biga}@ispras.ru
http://www.ispras.ru

Abstract. This paper presents an approach aimed at simplifying development of portable Linux applications, suggesting a method of detecting compatibility problems between any Linux application and distribution by means of static analysis of executable files and shared libraries.

In the paper we concern the idea of successful launching of application in a distribution. A formal model is constructed that describes interfaces invoked during the program launching. A set of conditions is derived that should be satisfied by application's and distribution's files in order to make it possible for application to successfully launch in distribution. The Linux Application Checker tool is described that supports the approach and allows to detect portability problems of applications at early stage of development.

Keywords: Software portability, Software maintenance, Linux.

1 Introduction

Nowadays hundreds of Linux distributions exist. Most of them are based on the same set of components – Linux kernel, GNU utilities and libraries, KDE or Gnome desktop environment, etc. Many of these components follow the "Release Early, Release Often" principle, and new versions can be released every month or even every week. As a result, different distributions provide different versions of the same components. Unfortunately, though changes between successive versions can be relatively small, it is not uncommon for developers to break backward compatibility. In addition, distribution vendors often modify original code of components in order to fix known issues, to increase performance or to add some features that would be unique for their distribution. Thus, there are a lot of differences in functionality of the same components in different Linux distributions. This significantly complicates the task of development of Linux applications that could be launched on as many Linux distributions as possible.

Developers of open source applications usually rely on distribution vendors – most distributions have a *maintainer* for every application, who is responsible for correct functionality of the application in the distribution and can modify its code, if necessary. However, application should be rather popular to be included in many distributions; and even for open source programs, increasing portability allows to save maintainers' efforts. Finally, sometimes modifications introduced by distribution developers are criticized by original application authors.

Developers of proprietary software cannot rely on distribution vendors, since they don't publish the code and only provide compiled binary files. In this case, it is application vendor who is responsible for proper functionality of software product on all supported platforms. However, large amount of existing Linux-based systems makes systematic testing of application on every platform quite expensive even for large vendors. Moreover, not only hundreds of different Linux distributions exist, but many Linuxes run on dozen of hardware architectures. Many hardware platforms are not broadly accessible, while they are still potentially interesting for application vendors (for example, IBM zSeries platform is even more interesting for some developers of large enterprise software then the 'usual' x86).

Very often impossibility of guarantying proper functionality of the product in many distributions leads to limitation of officially supported systems – usually to a very few number of distributions that have the major market share, such as SUSE Enterprise Linux (SLES) or Red Hat Enterprise Linux (RHEL). That is, vendors do not tend to cover the whole Linux market. But as for end users, they normally want to have applications for 'Linux', not for 'RHEL' or 'SLES'.

Thus, it is desired for application developers to be able to achieve compatibility of their product with as many Linux distributions as possible. Since manual testing of applications in every existing system is a very hard task, it is important to detect as many portability problems as possible without performing runtime testing. One of the possible approaches that can help here is static analysis of application binary files. In general, such analysis may not guarantee the full compatibility of application and distribution, but can allow developers to save their efforts by decreasing time of problem detection.

The remainder of the paper is structured as follows: Section 2 gives a review of existing approaches to allow Linux application developers to support as many distributions as possible. Section 3 introduces a formal approach to analysis of compatibility of any pair of application and distribution by means of static analysis of their executable files and shared libraries. Section 4 presents the Linux Application Checker tool that supports the suggested approach and allows to detect issues in compatibility between any given application and most popular Linux distributions. Finally, Section 5 summarizes the main ideas.

2 Existing Approaches

Importance of improving portability of Linux applications is realized by all members of the Linux community and all participants of the market – distribution vendors, developers of open source projects, independent vendors of proprietary software and end users. Naturally, initiatives and approaches to increasing application portability come from different categories of community members. In this section we consider the most popular ways for application vendors to achieve compatibility of their product with existing Linux distributions.

2.1 Using a Testing Farm

The most straightforward way for developers to guarantee compatibility of their program with some distribution is to thoroughly test the program in this distribution. In order to perform such testing, one should set up a separate machine (either physical or virtual), install the target distribution there, then install the program itself and run the tests. During the development process, developers should have a set of machines with target distributions, where their application is periodically built and subjected to testing. Such a set of machines is often referred to as a *testing farm*.

Large set of target platforms requires large set of machines to be set up. Even if machines are virtual, this can consume significant resources. Testing farms are usually served by automated scripts that schedule regularly builds, testing and other actions. Such infrastructure does not require much efforts for maintenance, but in general it increases the duration of 'detect problem – fix it – test again' chain (if compared to the case when developer is able to perform testing directly on his machine).

In addition, if developers want to cover hardware platform which is not broadly accessible, then they can find that time of real machines is expensive, while performance of emulators (if any) is usually poor.

Thus, though runtime testing in all target platforms is a really important thing, it is desired to detect as many problems as possible before running the tests inside the testing farm. It is especially important to detect critical issues whose presence will make the further testing useless.

2.2 OpenSUSE Build Service

An interesting initiative is provided by the OpenSUSE Build Service (OBS) – an infrastructure that can used by developers to build their applications for different distributions without direct access to the target systems. Nowadays, the OBS supports OpenSUSE, Mandriva, Fedora, Debian and Ubuntu; more systems to be added in future [4]. Though it is clear that periodic build of a large variety of applications requires significant resources, so it is not cheap to add a new target distribution.

As for portability, the service allows to handle several kinds of differences between distributions, such as package format or system file location. However, many aspects (e.g., differences in the behavior of the same function) are out of its scope. Surely, developers can integrate execution of automated tests in the build process – this will allow them to use OBS as a testing farm, whose disadvantages were discussed above.

Finally, the OBS approach makes sense only for developers of the open source products, since one have to provide application source code to the service.

2.3 Creating Standard-Compliant Applications

One more way to create a portable application is to focus on standards for operating systems that specify the set of interfaces that every compliant system should provide. The advantage of this approach is that standards not only guarantee the presence of interfaces, but also specifies all their characteristics that should be accessible by applications (e.g., behavior of functions). In order to be able to give such guarantees to its users, every standard usually provides a set of tests (usually referred to as *certification test suite*) that perform thorough testing of the standardized interfaces and should be passed by every implementation in order to confirm its compliance with the standard. Thus, if application uses only standard interfaces, it is guaranteed that application will demonstrate the same behavior in all standard-compliant distributions.

In the Linux world, the most famous standards are POSIX and Linux Standard Base (LSB), that concern Application Programming Interface (API) and Application Binary Interface (ABI) correspondingly. Both POSIX and LSB take into account existing systems (UNIX-like systems in case of POSIX and Linux in case of LSB) and try to standardize only those items that are implemented in all major distributions and proved to be mature, stable and useful. In those cases when some interface is provided by all systems but with slightly different characteristics, both POSIX and LSB try to specify only those aspects that are common for all implementations; relying on other aspects is not recommended for developers.

However, no standard can cover all possible interfaces – for example, POSIX only concerns the core system libraries (specifying about 1.000 of functions) and utilities. LSB covers more libraries, including some desktop and multimedia ones; however even its scope is still not broad enough for many programs – LSB 4.0 contains specifications for 57 libraries and about 38.000 functions, but a usual Linux distribution on a single DVD disc provides about several thousands of libraries and hundreds of thousands of functions [6]. Finally, trying to specify the 'common core' of existing systems often leaves many interface characteristics unspecified or declared to be 'implementation defined'.

Thus, among the existing methods described above, only the latter approach allows developers to avoid runtime testing of their applications in every target distribution. However, the main disadvantage of the approach (small coverage of existing interfaces) is hard to fix – standardization can be even more complicated and expensive task than development of tests, since creation of tests is usually only a part of the standardization process [10].

In this paper we suggest an approach that can be used to detect certain portability issues without runtime testing, allowing to decrease the time spent on problem detection. To detect portability problems, a lightweight static analysis of binary files of application and target distributions is used. The approach can be also used in cooperative with the standard-oriented method in order to analyze portability of those application parts that are not covered by existing standards.

3 Static Analysis of Interfaces Involved in Interaction between Distributions and Applications

Let *Distros* and *Apps* be the sets of all the Linux distributions and applications respectively. Below we will consider compatibility aspects of some application $A \in Apps$ and some distribution $D \in Distros$. First, let's clarify what do we mean under compatibility.

In general, interaction between two systems is performed by means of *interfaces* – one participant provides interfaces, the other uses them. For example, operating systems provides libraries that export functions, applications load these libraries and call their functions.

If distribution D provides a set of interfaces I_p and application A uses a set of interfaces I_r, then in order for successful interaction between A and D to be possible, the following condition is necessary:

$$I_r \subseteq I_p \tag{1}$$

This is a very general criterion – in every particular case one should point out concrete kinds of interfaces that should be taken into account. Provided that all possible kinds of interfaces and their properties are taken into account, this is also a sufficient condition.

For practical usage, it is important to have a way to analyze which interfaces are provided by distribution and which are used by application. Every interface may have a large set of different properties, all of which can be divided on two groups:

1. Properties that can be checked statically, without a need to invoke the interface (e.g., function signature).
2. Properties that require runtime testing (e.g., function behavior).

Surely, this classification is not ultimate – for example, one may claim that function behavior can be verified statically if its source code is available; and on the opposite side, even checking function signature in some situations may require function invocation (e.g., when there is no access to its declaration, but only to binary library that exports this function).

In this paper, we consider only those interfaces between Linux distributions and applications for which the condition 1 can be checked statically at a relatively low cost. Though the resulting set of interfaces is quite limited (in particular, it doesn't include any behavioral aspects), satisfaction of the condition 1 for this set guarantees that the application can be *successfully launched* in the distribution. To start with, let's clarify what we mean under these two words.

In our work, we consider *binary* applications – that is, applications that consist of binary executable files and shared libraries (*shared objects*, in Linux terminology). Any application in our model is a set of binary files, every of which is either a shared library or an executable file. Similarly, every Linux distribution is considered to be a set of shared libraries and binary executables.

Let's say that the application A *successfully launches* in distribution D, if dynamic loader in this distribution is able to form an executable image of the application in memory and pass the control to the application's entry point.

In Linux, for executable files and shared objects the Executable and Linking Format (ELF) is used, described in [1] and [2]. An ELF file can be self-sufficient in that sense that it may not require any external interfaces to be present in the system and work directly with hardware (maybe using low-level kernel interfaces, if direct access to hardware is not allowed). In this case dynamic loader just loads the file into memory and right after that passes control to its entry point; however, such files are used rarely and are not interesting for us.

Most programs nowadays use the advantage of dynamic linking, leaving the task of implementing routine functions for system libraries and concentrating only on unique features of their own. In this case dynamic loader has to perform much more actions to form a memory image for application. The precise algorithm is quite complex [5], but we are interested only in the following steps where the process can fail because distribution doesn't provide interfaces with required properties:

- Check that ELF files participating in dynamic linking have format acceptable for the loader – don't contain unknown ELF sections, have proper target hardware architecture, etc.
- Resolve dependencies on libraries – detect which shared libraries should be loaded to satisfy application's needs.
- Check that all dependencies on versions of binary symbols required for the loaded files are satisfied.
- Resolve addresses of external binary symbols of every loaded file – for every such symbol the actual implementation should be found among the loaded libraries.

After these tasks are complete, a memory image is constructed and dynamic loader passes control to the entry point of the file being launched. If this point is reached, we can say that the application has been successfully launched (in our terms).

Using the terms from the dynamic loading algorithm description, we can make our representation of Linux applications and distributions more accurate – every application and every distribution is considered as a set of ELF files, every of which can provide and require libraries, binary symbols and symbol versions. Using this representation, in the next sections we will derive a set of conditions that should be satisfied in order for the application A to be successfully launched in the distribution D. All these conditions can be checked statically by analyzing ELF files of application and distribution without a need to actual installation and launching. Moreover, there is no need to emulate the work of the dynamic loader – all conditions can be checked in a much more simple way and a lightweight analyzer can be developed to automate this process.

Now let's consider every kind of interfaces involved in the dynamic loading process.

3.1 ELF Sections

The ELF format evolves quite slowly (if compared to other parts of the Linux ecosystem). However, from time to time significant additions are introduced and files that use these additions cannot be used in older systems. The most notable example of the last years is introduction of the **.gnu.hash** section aimed to provide hashing with higher performance than the 'usual' **.hash** section. Introduced in 2006, this change led to the fact that programs compiled in new generation of distributions (such as RHEL 5 or Fedora 6) failed to run in previous releases of the same systems (RHEL 4, Fedora 5) [3].

Though such significant changes are introduced very rare, one should remember about them and check that dynamic loaders in target distributions support all aspects of the ELF format that are used in the application files. Thus, if $SysSupportedELF(D)$ is a set of ELF features supported by the distribution D, and $FileReqELF(f)$ is a set of ELF features used by the file f, then the following condition should be satisfied in order to launch application A in D:

$$\forall f \in A \rightarrow FileReqELF(f) \subseteq SysSupportedELF(D) \tag{2}$$

3.2 Shared Libraries

Let $SharedLibs$ be a set of all shared objects in the Linux ecosystem. Every library $lib \in SharedLibs$ is characterized by its *soname* (a special name visible to dynamic loader which may or may not be equal to the library file name) and hardware architecture: $lib = (soname, arch)$. It is the soname that is 'required' by ELF files; however, when looking for library that provides the requested soname, dynamic loader takes into account both library's soname and name of its file – if any of them matches the requested soname, than the loader picks the library up to satisfy the request. Thus, if soname of some library differs from its file name, then this library should be represented in the $SharedLibs$ set by two entities: $(soname, arch)$ and $(filename, arch)$. Let $FileProvLibs(f)$ to be a set of $SharedLibs$ elements provided by a file f (this set consists of either one or two elements).

Every ELF file can have a set of DT_NEEDED entries in its dynamic section that store sonames of libraries required by the file. Let's designate this set of required libraries as $FileReqLibs(f)$. This is a subset of our $SharedLibs$ set, with target hardware architecture of every $FileReqLibs(f)$ element been equal to target architecture of the file f itself (that can be detected on the basis of **Class** and **Machine** fields of the ELF header, as described in [6]).

For every distribution D, we are interested in the whole set of libraries provided by it, which is a union of libraries provided by all distribution files:

$$SysProvLibs(D) = \bigcup_{f \in D} FileProvLibs(f)$$

For applications, on the opposite, one should build a set of required libraries that are not provided by the application itself and thus are expected to be present in the system:

$$AppReqLibs(A) = \\ = \bigcup_{f \in A} FileReqLibs(f) \setminus \bigcup_{f \in A} FileProvLibs(f)$$

The second necessary condition that should be satisfied in order for the application A to be launched in the distribution D is that all libraries required by A should be provided by D:

$$AppReqLibs(A) \subseteq SysProvLibs(D) \tag{3}$$

3.3 Symbol Versions

A specific feature of ELF files in Linux is possibility of assigning a particular *version* to any binary symbol – different (from the source code point of view) functions or global variables can be made visible on the binary level under the same name but with different versions. This allows to keep backward compatibility with old applications when library developers decide to change function behavior – the old function implementation in this case becomes frozen and visible on the binary level with the same name as before. The new implementation is also visible under this name, but with a different version.

Every version is a simple literal string. Information about versions exported by file and versions required by it is stored in the appropriate ELF sections. When loading a file into memory, the system loader compares the set of required versions with versions provided by libraries loaded as file dependencies. Let's designate the latter set as $FileLoadedLibs(f, D)$; it is built using the following algorithm:

1. Set $FileLoadedLibs(f, D) = \emptyset$.
2. Put all $FileReqLibs(f)$ elements to $FileLoadedLibs(f, D)$ and to a temporary $AddedLibs$ set.
3. For each library $l \in AddedLibs$, calculate dependencies $FileReqLibs(l)$ of the ELF file that represents the library, calculate its difference with the $FileLoadedLibs(f, D)$ and union such differences to a new set:

$$FileIndirectDeps(f, D) = \\ = \bigcup_{l \in AddedLibs} FileReqLibs(l) \setminus FileLoadedLibs(f, D)$$

4. If $FileIndirectDeps(f, D)$ is not empty, then put all its elements to the $FileLoadedLibs(f, D)$ set.
   Rebuild $AddedLibs$ to be equal to $FileIndirectDeps(f, D)$.
   Set $FileIndirectDeps(f, D) = \emptyset$ and go to step 3.
5. Otherwise, everything is done and $FileLoadedLibs(f, D)$ is built.

Thus, the $FileLoadedLibs(f, D)$ set consists of libraries directly required by the file (that is, $FileReqLibs(f) \subseteq FileLoadedLibs(f, D)$), expanded with libraries recursively loaded as dependencies of these libraries in a particular distribution. Since dependencies of system libraries are specific to a particular distribution, the $FileLoadedLibs$ set for the same file f can be different on different systems. That's why we write that this set is a function of both file f and distribution D.

Let $SysProvVers(f, D)$ to be a union of versions provided by files from $FileLoadedLibs(f, D)$. With $FileReqVers(f)$ standing for versions required by the file f, we can formulate the following necessary condition that should be satisfied for the application A to be launched in the distribution D:

$$\forall f \in A \rightarrow FileReqVers(f) \subseteq SysProvVers(f, D) \tag{4}$$

3.4 Binary Symbols

If all previous checks are completed successfully, the dynamic loader proceeds with resolution of external binary symbols for the files been loaded. Every binary symbol is unambiguously identified by name and version: $s = (name, version)$. The resolution process is similar to the one for versions of binary symbols – the loader takes a set of binary symbols required by file ($FileReqSyms(f)$) and then compares it with $SysProvSyms(f, D)$ – a set of symbols provided by libraries from the $FileLoadedLibs(f, D)$ set. So one more necessary condition for the successful launch is like the following:

$$\forall f \in A \rightarrow FileReqSyms(f) \subseteq SysProvSyms(f, D) \tag{5}$$

Strictly speaking, there can be a situation that if $f \in A$ is a shared object, then it is never launched directly, but only loaded along with other files. In this case some symbols required by it can be provided not by libraries from the $SysProvSyms(f, D)$ set, but by libraries from $SysProvSyms$ sets for files that are loaded together with f, since finally all these files are joined to a single image. Build tools in Linux allow programmers to perform such tricks. However, dependencies of the same library in different systems can vary (and may change as time goes by), so these tricks are not considered to be a good practice, especially from portability point of view. In our work, we ignore such possibility and treat 5 as required condition.

Now let's consider the fact the sets of required and provided versions are constructed automatically during application or library build as unions of versions of required and provided symbols respectively. That is,

$$version \in FileReqVers(f) \Leftrightarrow \exists s = (name, version) \in FileReqSyms(f)$$

$$version \in SysProvVers(f) \Leftrightarrow \exists s = (name, version) \in SysProvSyms(f, D)$$

So if 5 is satisfied, then 4 is also satisfied, that is, 5 $\Rightarrow$ 4, and it is actually enough to only check the condition 5. However, in real systems a number of required symbols is usually much more greater than number of required versions, so 4 can be checked much more faster. Thus, it may still make sense to check 4 in order to detect possible problems at early stage, without deep analysis of binary symbols.

3.5 Sufficient Requirement

Up to this moment, we have considered the four necessary conditions 2, 3, 4 and 5 that should be met in order for the application A to be successfully launched in the distribution D. Since we have considered all interfaces involved in the launching process, then the sufficient condition of the successful launching is a conjunction of these conditions. As we have shown above, $5 \Rightarrow 4$, so the final sufficient condition can be formulated as a conjunction of 2, 3 and 5:

$$\begin{aligned} \forall f \in A \rightarrow \mathit{FileReqELF}(f) &\subseteq \mathit{SysSupportedELF}(D) \\ \mathit{AppReqLibs}(A) &\subseteq \mathit{SysProvLibs}(D) \\ \forall f \in A \rightarrow \mathit{FileReqSyms}(f) &\subseteq \mathit{SysProvSyms}(f, D) \end{aligned}$$

3.6 Method Value

It is clear that the approach suggested allows to detect only a limited set of compatibility problems; many kinds of issues (such as runtime function behavior) are out of its scope. In order to estimate the value of the approach in the real world, we have performed investigation of issue trackers of several Open Source projects and calculated percentage of issues that could be avoided if our approach were applied before the product release. We took into account errors concerning missing libraries and symbols; errors concerning symbol versions are relatively rare so they are joined with other symbol-related issues in the 'Failed symbols' column. Finally, it was found that issues related to the ELF format are very rare, so we haven't included them in the table below.

For our analysis, we have selected three popular applications that are broadly used in all Linux distributions – OpenOffice.org, Firefox and MySQL. In addition, we have investigated issues reported in the Launchpad software portal that provides hosting for more than 18,000 of Open Source projects. In our research, we have considered only critical bugs (either issues with severity set to 'Critical' or 'Blocker', or issues with the highest priority). Results of the analysis are shown in Table 1. The 'Total issue' column contains number of all critical bugs reported for the project; it would be also useful to calculate total number of bugs that concern portabilty, but this will require detailed investigation of every issue and is too time-consuming, so we haven't gathered such statistics.

Table 1. Number of issues in different projects that could be detected using the approach

Product	Total issues	Failed symbols	Failed libraries	Percentage
OpenOffice.org	20,000	190	110	1.5
Firefox	19,000	254	114	1.9
MySQL	6,400	98	20	1.8
Launchpad	42,000	95	60	0.3

It is clear that large applications that are actively used on almost all Linux distributions have greater percentage then relatively small projects hosted at the Launchpad whose target audience is, in general, much more smaller. Some projects from the Launchpad are not the binary ones, but created using interpreted languages (such as Perl or Python). For such programs, our approach is not applicable. Also note that our investigation only concerned bugs detected by customers, not by developers – that is, these are primarily bugs in the released products missed by QA teams.

Thus, using the approach suggested, large applications could decrease the number of critical issues discovered by customers by 1-2%. At the first glance, this is not a very high number, but it can be achieved at a very low cost – for example, the Linux Application Checker tool described below can perform all necessary analysis fully automatically, so developers will only have to spend several minutes launching the tool and checking the reports.

4 Linux Application Checker

The approach presented in this paper is implemented into an automated tool called Linux Application Checker. The tool can be used to analyze application binary files (in a form which is usually distributed to users) and check conditions 2, 3 and 5 for application and every distribution known to the tool. For every distribution, a verdict is given if application can be successfully launched there, and if not, a detailed list of compatibility problems is provided.

Data about distributions is collected by separate automated tools as described in [6] and is shipped with the tool, so the set of supported distributions is fixed for every tool version. The data collection process is fully automated and doesn't require for distribution to be installed – the analysis is performed on the basis of distribution installation packages (RPM and Deb packages are supported). In order to collect information about binary symbols and their versions exported by distribution libraries, these libraries are analyzed using 'readelf' and other tools from GNU Binary Utilities [7]. The same tools are used by the Application Checker to analyze application binary files on the user side.

The set of supported distributions is constantly updated; as of April, 2010, the tool contains knowledge about 63 distributions on Intel x86 architecture and 51 systems on x86-64 one (also known as AMD64). The tool supports five more hardware architectures: PowerPC, PowerPC64, IA64 (Itanium), S390 and S390X (IBM zSeries); for every of these platforms, knowledge about two dozens of distributions is collected (this number is smaller than for x86 and x86-64 due to the fact that many Linux variations don't support these platforms).

Finally, since the data collection process is fully automated and all necessary tools (including the Application Checker itself) are open, it is not hard for users to add data about any distribution they like.

It is important to note that Application Checker contains knowledge not about all libraries in every distribution, but only about a limited set of them (about 1,500 for every system). The thing is that according to empirical studies, there are a lot of libraries in the Linux ecosystem that are used very rarely (usually only by their developers), so it is unlikely that they will be required by any third-party applications [9].

In addition to data necessary to check the conditions 2, 3 and 5, the Application Checker also contains information about libraries and functions included in the latest version of the LSB specification. For LSB-compliant distributions, these libraries and functions are guaranteed not only to be present in the system, but to provide all the functionality required by LSB. Thus, if application under analysis uses only interfaces included in LSB, developers can be sure that the application will not only successfully launch, but will also be able to work properly in all distributions compliant with LSB. For symbols and libraries that were once considered as LSB candidates but were rejected by the LSB workgroup (usually due to their deprecated status or known bugs in the existing implementations), the tool provides developers with appropriate warning messages, suggesting an alternative, if possible.

While a positive result from the Application Checker does not guarantee that application will run correctly in all distributions, it can notably reduce the porting and testing costs; the tool it is easy to use, doesn't require much user actions and still able to detect a noticeable set of errors. Data from the LSB knowledge base makes the tool even more valuable, allowing people to combine standard-oriented development with the approach suggested in this paper.

Nowadays, Linux Application Checker is recommended by the Linux Foundation [8] to all software developers who want to improve cross-distribution portability of their applications. Moreover, it is possible for application vendors to apply for LSB certification using Application Checker reports – the tool has possibility to automatically submit reports to the LSB Certification System.

5 Conclusion

Existence of Linux applications that can be used in any Linux distribution without modifications are greeted with applause by all members of the Linux community – distribution vendors (who don't want to maintain huge sets of patches for every application in their systems), independent software vendors (who would like to support as many platforms as possible) and end users (who don't want to be bound to a particular distribution just because their favorite software doesn't support the other systems). However, large variety of existing distributions makes it hard to develop and to support a product that would run everywhere, especially for proprietary vendors who cannot rely on any third parties when solving portability problems.

Development of products compliant with some standard may help in some respect, but existing standards cover only a small piece of the Linux ecosystem. So systematic runtime testing on every target platform remains the most popular approach for guarantying compatibility, but in case of large variety of platforms it becomes too expensive.

In this paper, we have suggested an approach to detect portability problems of applications by means of lightweight static analysis, without a need for runtime tests. In general, the approach doesn't guarantee full compatibility of application and distribution, but it allows to check that the application at least can be *successfully launched* in the distribution. Problems concerning impossibility of launching of application are not rare in the real world.

The approach is based on the formal model of interfaces involved in the process of application launching. In the paper we use formalization to derive unambiguous conditions (both necessary and sufficient) of this process. Existence of clearly formulated and unambiguous requirements allows to create automated tools that can check these requirements.

The Linux Application Checker tool combines two techniques of increasing application portability without runtime testing. First, it allows to analyse program compatibility with different distributions using the approach described in this paper. Next, it supports development of applications that meet requirements of the Linux Standard Base specification by checking application compliance with LSB and suggesting alternatives for libraries and functions that are not included in the standard.

In our work, we have considered only programs represented as binary executable files and shared objects in ELF format. Applications that are written using interpreted languages (such as Java, Perl or Python) are beyond the scope of this paper. The approach can be extended to cover interpreted languages, too, since they also usually have some analogues of libraries exporting sets of functions that are used by applications. However, the concept of *successful launch* is not so clear for such applications, since there is no analogue of dynamic loader that resolves all external dependencies before passing control to the application itself. It can appear that even if a system doesn't provide all necessary interfaces, the application still can function correctly inside it until the missing interface is invoked. Nevertheless, it can still be desired to require that all interfaces that can be potentially invoked by the application should be present in the system. Our approach can be extended to cover this area, but derivation of formal necessary and sufficient conditions of compatibility will require investigation of interaction between application and interpreter, and this interaction can be actually specific to every particular interpreter language.

References

1. System V Application Binary Interface Draft (April 24, 2001), `http://refspecs.linuxfoundation.org/elf/gabi4+/contents.html`
2. Linux Standard Base Core Specification 4.0. Executable And Linking Format (ELF), `http://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic/elf-generic.html`
3. Proffitt, B.: More Compatibility Issues Easily Managed With LSB. Linux Developer Network (October 2008), `http://ldn.linuxfoundation.org/node/7141`
4. Schroter, A.: OpenSUSE.org Build Service – a Short Introduction. In: Free and Open Source Software Developers' European Meeting, FOSDEM (2008), `http://files.opensuse.org/opensuse/en/2/21/FOSDEM2008-OBS-short-introduction.pdf`

5. Drepper, U.: How To Write Shared Libraries. Red Hat, Inc. (August 20, 2006), http://people.redhat.com/drepper/dsohowto.pdf
6. Silakov, D.: Linux Distributions and Applications Analysis During Linux Standard Base Development. In: Proceedings of the Second Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE 2008), St.Petersburg, Russia, May 29-30, vol. 1, pp. 11–18 (2008)
7. GNU Binary Utilities, http://sourceware.org/binutils/docs/binutils/
8. The Linux Foundation: Building a Portable Application For Linux, http://ldn.linuxfoundation.org/lsb/check-your-app
9. Rubanov, V.: Automatic Analysis of Applications for Portability Across Linux Distributions. In: Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2009), York, United Kingdom, March 28, pp. 44–53 (2009)
10. Khoroshilov, A.V., Rubanov, V.V., Shatokhin, E.A.: Automated Formal Testing of C API Using T2C Framework. In: Proceedings of the Third International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008), Part 3, Porto Sani, Greece, October 13-15, pp. 56–70 (2008)