Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000

Library Interface Versioning in *Solaris* and *Linux*

David J. Brown and Karl Runge

Solaris Engineering, Sun Microsystems Inc.

Abstract

Shared libraries in *Solaris* and *Linux* use a versioning technique which allows the link editor to record an application's dependency on a particular release level of the library. The versioning mechanism operates at the level of the library's GLOBAL symbol names—a finer granularity than simply associating a version number with the library itself.

In *Solaris*, this mechanism has also been used to provide a means for the system-supplied shared libraries to define their application interface: to declare specifically which of their symbols are intended for application use (and are stable from one release to the next), and which are internal to the system's implementation (and hence subject to incompatible change).

This paper describes the library symbol-versioning technology in *Linux* and *Solaris*, the ways in which it is used to support upward compatibility for existing compiled applications from one release of *Solaris* to the next, and the potential for similar mechanisms to be applied in *Linux* versioned shared libraries.

1 Introduction

As *Linux* continues to grow in popularity, and more people come to depend on it, issues regarding compatibility will likely become increasingly important. In what follows we focus on a method for successive releases of system software, such as *Linux*[1] or *Solaris*, to maintain binary compatibility with existing applications. We describe some practices that are presently being used in *Solaris* (at the level of the system's library interfaces) to define and maintain an Application Binary Interface (ABI). We further describe how this definition along with an approach to library interface versioning helps to ensure the *stability* of existing application binaries across successive *Solaris* releases.

The hope is that some, if not all, of these practices can be incorporated into the development practices for *Linux* libraries to help increase binary stability for *Linux* applications. Due to the differences in development models between GNU/*Linux* and *Solaris*, some aspects of these schemes may need to be modified suitably to be beneficial to *Linux* development.

2 The ABI—a basis for system interface definition

Maintaining source-level or API compatibility, is well understood; less familiar is the idea of maintaining binary-level, or ABI, compatibility. The *Solaris* ABI (application binary interface) is the set of *runtime* interfaces in *Solaris* that may be depended upon by an application; if the ABI evolves in an upward-compatible way from one release of *Solaris* to the next, then existing compiled applications built on a given release will run on all subsequent releases without change (i.e. there is no fear that the application will break when run on a later release of the system).

The enormous value of access to source code may obscure the fact that there are many end-users who cannot exploit source-level compatibility and "simply" recompile their applications as needed when the system software has changed. Most often it is large organizations, as compared to individual users or developers, who are in this situation. The need for the new system's binary compatibility with their existing applications *may* be because they do not have access to the source code for certain applications, but much more often the logistical nightmare of recompiling, retesting and redistributing the hundreds or even thousands of applications that a typical large organization relies on, is the predominant issue.

Consequently, establishing a clear runtime interface for applications, and maintaining its binary compatibility is

1. When we use the term *Linux* in this paper, we are using the common shorthand, and mean a complete software system based on a Linux kernel (those provided by RedHat, SuSE, or Debian GNU/Linux are examples).

an important requirement for a supplier of operating system software. The means of satisfying that requirement is through precise definition of the system's application binary interface (ABI): If the producer of system software maintains the integrity of those interfaces, and the developers of software products which rely on the system also abide by it (and only use interfaces belonging to the ABI) then compatibility is assured.

2.1 Defining the *Solaris* ABI

At Sun, we tend to think of the system (*Solaris*) as the provider of a set of services, and these services are primarily provided to application programs and/or other layered software products, which are built and released independently to *Solaris*.

To a first approximation, the broader set of system services are provided to applications by the set of libraries supplied as part of the system. While the kernel is the underlying foundation for this, as provider of the most fundamental services (and there can be a great deal of focus on this, especially in the *Linux* environment), a complete software system represents a good deal more than the kernel alone: So in *Solaris* it is the set of system libraries, and more specifically the particular interfaces offered by those libraries, that we use to characterize the application's runtime interface.

Importantly, these interfaces are accessed by a *compiled* software product (such as an application) via *dynamic* linking—bindings made at *runtime* between the application and the shared objects which implement the system libraries. The use of shared libraries is critically important because dynamic linking to these libraries allows us to maintain a clear separation between a compiled application and the system implementation. Focusing on the runtime binding interface between the two is analogous to defining a protocol, or may be thought of as characterizing the interface to the *Solaris* virtual machine.

2.2 Defining system-internal interfaces

While the system libraries provide a set of interfaces to allow applications access to system services, most system libraries also expose some implementation interface—making visible a broader set of GLOBAL symbols than just those intended for use by applications. System libraries which are "lower" in the system's implementation-architecture hierarchy can have intimate interdependencies, so these libraries also expose some system-internal implementation interfaces. These interfaces, such as those needed in `libc` to support particular semantics in conjunction with `libthread` or `libnsl`, are really part of the system's *implementation* (as opposed to its external interface), and not intended for application use. To distinguish these two classes of interface, we define the following terms:

"Public" - interfaces which *are* intended for use by applications (and/or any other layered software products released asynchronously to *Solaris*), and which therefore have the upward compatible evolution property;

and "Private" - interfaces which are *not* intended for use by applications (or any asynchronously released layered software component), because these interfaces are part of the internal implementation of *Solaris*, and do not have the upward compatible evolution property.

In the following sections, where we discuss how change is managed, we will be focusing on changes to the Public interface—namely, those system interfaces that affect applications (or any other layered software product released *asynchronously* to the system software). In *Solaris* we have not yet attempted to provide for the asynchronous release of individual system libraries, such as might be addressed by carefully managing change to the interface between system libraries (i.e. the Private, or internal implementation interface manifest by the libraries). The assumption at present, therefore, is that a release of the system constitutes (and requires) a *synchronous* release of all the system libraries along with the kernel.

3 Versioning

As a system evolves, whether by the addition of new functionality, or via changes to the system's implementation of existing functionality (as are frequently done to improve its quality or performance), there is the need to indicate the kind of change. This is important, because applications, and other software products have been constructed which depend upon this functionality. Since changes to the system's interfaces can affect existing applications, some means to indicate the nature of the changes made is highly desirable.

A particular property that we consider important, and that we're trying to realize, is that developers of applications (and many other layered software products) can be insulated from changes made to the system they rely on for many of the kinds of change to that system's software.

3.1 Kinds of change—"major", "minor" and "micro" releases

At the outset is important to define the kinds of change that can be introduced to a system software product (and which also apply at a finer granularity to individual components of a system software product). In this paper we will focus specifically on the system's libraries, since libraries are the primary components of the system software which provide interfaces to other software products built to run on that system. The following three terms define a simple taxonomy of change which is applicable to all systems:

A *major* release is an *incompatible* change to the system software, and implies that [some] applications dependent on the earlier major release (specifically those that relied upon the specific features that have changed incompatibly) will need to be changed in order to work on the new major release.

A *minor* release of the system software is an *upward-compatible* change—one which adds some new interfaces, but maintains compatibility for all existing interfaces. Applications (or other software products) dependent on an earlier minor release will *not* need to be changed in order to work on the new minor release: Since the later release contains all the earlier interfaces, the change(s) imparted to the system does not affect those applications.

A *micro* release is a compatible change which does *not* add any new interfaces: A change is made to the *implementation* (such as to improve performance, scalability or some other qualitative property) but provides an interface *equivalent* to all other micro revisions at the same minor level. Again, dependent applications (or other software products) will *not* need to be changed in order to work on that release as the change imparted to the system (or library) does not undermine their dependencies.

3.2 Managing changes to the system interface

Now let's take a look at how the various kinds of change to system interfaces described above have been managed in some systems historically.

Virtually all systems that we are aware of incorporate some sort of version number in the filename of each library, and most systems that we are aware of have had some means of recording both the major and minor release concepts on a per-library basis in order to indicate and manage upward compatibility. Historically, several systems have recorded minor and/or micro release levels explicitly in the library's filename: For example, in Sequent's *Dynix* [Huiz 97] and Sun's *SunOS 4.x* (*Solaris's* OS component prior to *Solaris2*), library filenames were of the sort:

`lib`<*name*>`.so.`<*major*>.<*minor*>

(for example, `libc.so.2.9`), and in *Linux*, library filenames often also contain a micro (or "release") number, being of the sort:

`lib`<*name*>`.so.`<*major*>.<*minor*>.<*rls*>

(for example, `libz.1.1.3`).

3.2.1 Library minor release

When a library evolves compatibly, existing interfaces are preserved, but new ones are added (as needed for example, when new functional content is added). On many systems this change is reflected in the library (or libraries) affected by exhibiting both major and minor version numbers in the library's filename (e.g., `libfoo.so.1.2`) and incrementing the library's minor number to indicate that new interfaces were added (e.g. to `libfoo.so.1.3`). Since nothing has been done that would break applications constructed earlier (i.e. those that were built using either `libfoo.so.1.1` or `libfoo.so.1.2`), it is OK for these older applications to be linked with the newer library at run-time (although we have yet to describe how the application's dependencies are recorded and how this is managed at runtime, but will do so shortly).

3.2.2 Library major release

If the interfaces in a library shared object change incompatibly, not only must there be a way of indicating that a different version of the library has been created, but also of detecting application binaries that were built using the earlier edition of the library and preventing them from linking at runtime with the library's new major release.

Analogous to a library's minor release, the system or library provider's practice is to update the major revision number associated with the library. So for example, an incompatible change to `libfoo` would require that the successor to `libfoo.so.1` be named `libfoo.so.2`, or in the case that both major and minor numbers are reflected in the library filename (as in our preceding example), the naming revision would, for example, be from `libfoo.so.1.3` to `libfoo.so.2.1` (where 2.1 indicates the first minor release of major release 2).

3.3 How applications record library dependencies

Now while updating the major and/or minor version number on a library is a simple way for the system software or library provider to indicate the change, this practice presumes of course, that there is also some mechanism for labelling application binaries with the revision levels of the libraries they have used. And at run-time, some mechanism must be present to ensure a rendezvous of the application with an appropriate version of each library it requires.

There must, at the very least, be a way of marking applications with the name and *major* revision level of a library they were built with to ensure that application executables requiring a given major version of a library (e.g. `libfoo.so.2`) are not accidently linked with another major version of the library, either a later (`libfoo.so.3`) or earlier one (`libfoo.so.1`), since we know that these are incompatible with the application. Related to this is that later releases of a system software product, if they are to be able to run older application binaries, must continue to provide earlier major edition(s) of the libraries, and have a mechanism for the older applications to be linked at runtime to the major version they require.

3.3.1 Historical Practice

On some earlier systems, when an application was built, the link editor simply recorded within the application binary, the *filename* of each library that it depended upon (that is, a name containing the major and minor number of the library present on the build platform).

At run time, upward compatibility could be handled by the runtime linker's explicit knowledge of the semantics of the version numbers contained in the library filenames. In SunOS 4.x for example, the runtime linker would look for a library with the same name and same *major* release number as that recorded as a dependency in the application (and in the presence of multiple minor-version instances of the library that match the major number, the instance with the highest minor version number is used) [Gingell 87].

If the minor version number of the library found on the runtime platform was greater or equal to that of the dependency recorded in the application binary, dynamic linking proceeded silently (since the library must contain all the content required). In the case that the minor revision found on the runtime platform was lower than that recorded in the binary SunOS 4.x had the policy of issuing a warning diagnostic (that an earlier version had been found), but allowing the runtime linking to proceed. The application *might* still run successfully if it had only happened to depend on functional content (and interfaces) present in the earlier release of the library. If the application had depended on later content however, a runtime relocation error would occur when the application invoked an interface not present in the library on the runtime platform. Other systems may have adopted a more conservative policy and disallowed an application to run if a minor release of at least equal minor revision level to that required by the application was present on the runtime platform.

Neither of these policies is particularly satisfactory however, as the former might permit an application to run whose dependencies can't be met, while the latter prevents a class of applications whose interface dependencies could be met (i.e. whose interface needs were limited to content present in an earlier minor edition of the library, but were built on platforms with later minor editions), from being allowed to run on platforms bearing an earlier edition of the library than the application was built (and hence labelled) with, but which in fact happened to provide everything the application actually required.

3.4 The minor-version rendezvous problem

The important but somewhat subtle issue associated with *minor* revision changes, just described, is that an application built with a given minor release of a library *might*, but *cannot be certain* to run on an earlier minor release level of the library. This is because the application may have used one or more of the interfaces added to the library at a later minor release level, and not present in the earlier one.

To resolve this problem we must know more than just what minor release level an application was built with. We must know more specifically what content within that library it is actually *dependent upon* if we are to determine whether that content is present in the library found on a particular runtime system that the application is being run against. This is one of the key concerns addressed by the library versioning and linking technology present in both *Solaris* and *Linux*-based systems[2].

4 Library versioning in *Solaris* and *Linux*

Many contemporary systems (including *Solaris* and *Linux*-based systems) use the ELF object file format (the SystemV Executable and Linking Format) [SysVABI]. Where ELF is used, dynamically-linked libraries (libraries implemented as shared objects) contain an *so-name*—a specific means of naming the library (superceding the library's filename) stored within the library's object file[3].

When an application (or other dynamic object) which depends on the library is built, it is the library's *so-name* (not the filename) that is recorded in the application binary as a dependency.[4] And when the application is

2. Those using GNU libc version 2 or later
3. The DT_SONAME contained in the shared object's .dynamic section.

run, dependency information contained in the application binary is used by the runtime linker to locate and load the libraries depended upon by the application.

In order to allow for upward compatible evolution of the library—to permit an application constructed on a given system release to encounter a different minor revision of the library on the runtime platform (and still run successfully), current practice is to have the library's *so-name* contain only the major number (e.g. `libc.so.1`). At this level applications record a dependency only on the particular name and *major* release of the library and may be run with any minor release level they encounter on a runtime platform (albeit with the expectation of finding a minor release level sufficient to provide the interface content they depend upon).

For minor versioning, rather than the traditional method of associating a single minor version number with the library and incrementing it (as a way of saying "something was added"), a more useful approach is to define specifically *what* has been added, and to record this information in the library shared object itself.

So, instead of simply renaming the library (e.g. from `libfoo.so.1.2` to `libfoo.so.1.3`), the library's name remains `libfoo.so.1` (reflecting its major release level) and inside the library, a label is introduced (say, "VERS_1.3") that indicated the GLOBAL symbols added in the third minor revision level. If such labels are added with each minor revision level (e.g. VERS_1.1, VERS_1.2, VERS_1.3, ...) and all earlier labels preserved within subsequent minor release editions of the library, the evolution of the library's interface can be seen clearly.

4.1 Basic mechanism

In 1995 the *Solaris* 2.5 link editor (`ld`) and the run-time linker (`ld.so.1`) were enhanced to support "versioning" and "scoping" of symbols in shared objects.

The versioning mechanism itself is fairly stratightforward, simply allowing for the definition of named sets, each of which contains a specified list of symbols. Sets may be defined by providing an explicit list of symbols, and/or by referring to other sets by their name to include (inherit) the symbols in those sets[5] [Solaris LLM].

4. More specifically the *so-name* is recorded as a DT_NEEDED entry in the application object's .dynamic section for each library used by the application.
5. Full details of the library versioning technology in Solaris may be found in the *Solaris Linker and Libraries Manual* in the section describing "Versioning".

The GNU/*Linux* implementation[6] is the same as that in *Solaris*, although the GNU implementation provides two extensions [GNU_ld]: First, as an alternative to providing definitions in a separate mapfile, "`.symver`" assembler directives may be included in-line in the C source code for the library. Second, a form of interface overloading is provided: Multiple (incompatible) versions of the same function are allowed to exist in a single revision of the library. This is done by mapping an external symbol name (as referred to by an application) to a different internal name for the function, depending on the (minor) version set specified by the application's dependency. Special `.symver` directives must be provided to indicate the per-version mapping:

```
.symver old_printf, printf@VERS_1.1
.symver new_printf, printf@VERS_2.0
```

The intention is to allow several incompatible versions of any individual interface to be carried simultaneously within the library, and thus not have to increase the library's major version number[7].

When a *versioned* library (shared object) is built, a "mapfile[8]" containing the list of exported symbols grouped into named sets[9] is passed to the link editor `ld(1)` via the `-M` option.

As a simple example, consider the versioning mapfile for a hypothetical library named `libstack.so.1`:

```
SUNW_1.1 {
    global:
        pop;
        push;
}

SUNWprivate {
    global:
        __pop;
        __push;
    local:
        *;
}
```

6. The mechanics for symbol versioning in *Linux* are implemented in the GNU link editor (`ld`), provided in the `binutils` package
7. We are not yet sure of the broader policies describing the expected use of this mechanism however.
8. What is called a "versioning mapfile" in Solaris is called a "VERSION script" by the GNU link editor, but their syntax is the same.
9. These named sets are called "versions" in the parlance of the Solaris Linker and Libraries Manual, indicating the primary purpose intended by their design.

In *Solaris* system libraries, by convention, the "SUNW" prefix[10] is used in the set names (versions).

The versioning mapfile above instructs the link editor (`ld(1)`) to construct a shared object which exports (as GLOBALs) the symbols `pop`, `push`, `__pop`, and `__push` for use by other binary objects (executables or other shared objects). The "`local: *;`" directive instructs the link editor to take all remaining GLOBAL symbols defined in the objects being linked and make them inaccessible external to the shared object being produced: Essentially the link editor "demotes" these symbols from GLOBAL to LOCAL symbols (see 4.1.2 "Scoping" below). For example, utility functions that are part of the internal implementation interface of the library (and hence intended only for use within the shared object) will not be exported.

Suppose that in a later revision of `libstack.so.1` it is decided that a `swap()` functionality is desired, then the mapfile would be the same as above, but with an additional version definition:

```
SUNW_1.2 {
    global:
        swap;
} SUNW_1.1;
...
```

This notation reflects the upward-compatible evolution of the library's Public interface, in which the set SUNW_1.2 defines two new interfaces, and inherits those interfaces in the set named SUNW_1.1. The inheritance chain of symbol sets SUNW_1.1 .. SUNW_1.2 ..., and so on, evolves corresponding to each new revision that adds interfaces to the library. Note that the version numbering scheme following the SUNW prefix is a major and minor number pair, where the major number corresponds to the major revision number of the library.

4.1.1 Versioning

The immediate effect of the library's versioning is that at the time an application is built (compiled and linked), the link editor can record into the application binary the names of any versions (named sets of symbols) in the library that the application depends on. This is the default build practice for *Solaris* applications. Importantly, it is not the name of the *latest* set (version) present in the library that is recorded, but the smallest set (or sets) containing those symbols *depended upon* by the application: For example, if `libc.so.1` contained six minor revision levels, of which the latest was `SUNW_1.6` on the platform used to build `test_app`, but this application only relied on symbols present in revisions up to the third minor release (`SUNW_1.3`), then the application would be labelled with that named set to indicate its correct minor version dependency.

This permits applications built on later minor release editions of the library to be run (correctly) with earlier editions of the library, when their interface requirements are constrained to an earlier release level. And second, it ensures that applications which record a dependency on a given named set (minor revision level) will not be run[11] with an edition of the library which does not possess that named set.

When the application is run, the runtime linker uses the version dependency information recorded in the application binary to determine if all these named sets (interfaces required by the application), are present in the library found on the runtime platform. This ensures that the sufficient *minor* revision content is present in the library to meet the application's needs (thus going beyond simply using the application's list of NEEDED *so-names* to locate the correct *major* versions of the libraries).

4.1.2 Scoping

Somewhat specifically designed to overcome a shortcoming of the C language's symbol scoping capabilities, implementation interface which is used *only* internal to a library itself (i.e. interface used only within a single dynamic object) can be handled specially. A capability is afforded by the *Solaris* link editor which permits a reduction in the scope of those interfaces from GLOBAL to LOCAL within the library at the time that the library (dynamic object) is linked. We refer to this as "scope reduction" or library-level "scoping" of symbols.

The keyword `local:` in a mapfile is a scope-reduction directive, and provides that one or more symbols intended for use only within the shared object itself may be treated as library-level STATIC symbols. In this way the shared library can control what symbols are intended for export. Scope-reduced symbols are changed from library GLOBAL symbols to LOCAL symbols as part of the

10. Originally this was intended as a way to distinguish interfaces introduced by Sun Microsystems—and therefore perhaps particular to Solaris, from those defined by broader standards, such as the SystemV ABI or the Open Group's UNIX) . The prefix "GLIBC" is used by the GNU/glibc package in a similar convention.

11. At application start-up, a warning is emitted by the Solaris runtime linker indicating that the library does not contain the version required by the application, and the application exits. This is in lieu of a runtime relocation error if the application were allowed to execute.

link editing process which produces the shared object, thus preventing application programs (or any other dynamic object) from accidentally (or intentionally) using them. As a corresponding effect, these symbols are removed from the dynamic symbol table. Note that scope-reduced symbols are not actually associated with any named set.

4.2 Versioning practices (policies) in *Solaris*

To implement *Solaris's* interface definition and upward compatibility policies, we have defined a set of practices which apply the library versioning mechanisms described above. These practices are now used at Sun as an intrinsic part of *Solaris*'s library development practices.

Sun defines the *Solaris* ABI in terms of the interfaces to the system's libraries. First, at the library-naming level, libraries are given a filename and *so-name* corresponding to the library's major release level. Minor versioning information is contained within the library binary.

In order to make clear which of a library's GLOBAL symbols are part of the ABI and which aren't, the symbol versioning mechanism described above are used to classify all GLOBAL symbols (as Public or Private, and by ascribing each Public interface to a set indicating the minor release level of the library in which it was introduced). The set of all symbols indicated to be Public in a system library constitutes the library's ABI, and the collection of all such libraries in a given release of *Solaris* thus constitutes the *Solaris* ABI. This ABI is self-documenting since the definitional information (which symbols are Public and which are Private) is part of the library itself[12] and readily accessible through system utilities such as the `pvs(1)` ("print version section") command.

When the library versioning mechanisms were first applied to the *Solaris* shared libraries, the scoping mechanism was applied to hide all interfaces that are part of the linkage between the individual compilation units (`.o` files) of the library, but used *only* within the library itself. These symbols are "scoped out" (demoted from GLOBAL to LOCAL in the link-editor's construction of the shared object), so that these symbols are not visible external to the library and cannot be used by any external dynamic object.

Remaining GLOBAL symbols (those interfaces that must be visible external to the library) are separated into Public and Private. GLOBAL symbols classified as Public name interfaces intended for use by application developers (they are documented and guaranteed not to change incompatibly from one release of *Solaris* to the next). Private symbols name interfaces that are part of the *Solaris* implementation (they can not be guaranteed to remain compatible, or even to persist at all, from one Solaris release to the next, and are not suitable for use by application developers).

To reflect the upward compatible evolution represented by a series of minor revisions to the library, the Public symbols appear as a number of named sets of the form: "SUNW_<*major*>.<*minor*>". Each named set (version) identifies the full interface content present in a given minor revision of the library. The set lists the Public symbols introduced in that minor release, and names its predecessor to inherit its contents (e.g. SUNW_1.2 explicitly identifies the set of symbols added in the second minor release of `libc`, and inherits SUNW_1.1—the set of symbols present prior to that). A new version is added to the library only when a release of the library introduces new interface content.

All Private symbols, in contrast, are associated with a single version named "SUNWprivate". Symbols may be added to (or removed from) this set from one release to the next, and since there is no expectation of upward compatibility in this set there is no inheritance chain of versions for Private symbols. Recall that Private means (system-internal implementation interface) and that applications must not depend on these symbols. The contents (or even the existence) of this set therefore should not matter to an application.

All of the system libraries in *Solaris* which provide the basic OS and core networking services, as well as many of the basic window system interfaces, have been versioned in this way since *Solaris 2.6*. The eventual goal is to version all libraries shipped by Sun which can be used with *Solaris*. In due course it is hoped that the same approach will be taken by libraries built by other developers—particularly those "middleware" products which are not included with the *Solaris* release, where such libraries offer application-usable interfaces. The intent is that all layered products that can be used with

12. These definitions are contained within each library binary. They are reflected within the shared object by three ELF sections: Two sections named .SUNW_version and one named .SUNW_versym. The first has sh_type: SHT_SUNW_verdef, and gives all those versions (named sets of symbols) defined by the library. The second section has sh_type: SHT_SUNW_verneed and lists versions (named sets of symbols in other shared objects) depended upon by the library). The third has sh_type: SHT_SUNW_versym, and associates a set of GLOBAL symbols in the library with a respective "version" (named set) listed in the first section in order to define each such set name.

Solaris define *stable* application interfaces, in order to realize similar benefits of upward binary compatibility for applications that depend upon them.

4.3 Versioning practices in *Linux*-based systems

The GNU “glibc” package provides about 20 shared libraries (including `libc`) and makes extensive use of the versioning mechanism in `ld(1)`, both to implement scope reduction for library-internal symbols, and to indicate the library’s minor release evolution through versioning. For example, in `libc.so` the current version chain is:

GLIBC_2.0, GLIBC_2.1, GLIBC_2.1.1, GLIBC_2.1.2

For libraries that have not been added to recently, the highest version remains the last one in which content was added. For example, the highest version in `libcrypt.so` is GLIBC_2.0. If a new library is added at a certain version of the GNU glibc package its initial version set name is that of the corresponding package: e.g. `librt.so` begins at GLIBC_2.1

Looking at the Redhat Linux 6.2 release, one can see that most of the libraries that are not part of the glibc package (e.g. those of `XFree86` and `libgtk`) are not as well managed: While most have versions defined in them, these are currently only a default version with no structure yet defined (that is, there are only two versions `lib`<*name*>.so.<*n*> and `GCC.INTERNAL`). These libraries currently have no Public inheritance set chain defined.

The most important difference between GNU/*Linux* and *Solaris* is that the GNU `glibc` libraries do not distinguish the system’s internal implementation-interface[13] from their application interface. By making it clear that application developers should not use implementation interfaces (see section 5), *Solaris* library developers can change the library’s *implementation* in the future (for example, to substitute new algorithms or to achieve performance gains within the Solaris system libraries), without the fear that existing applications could be broken.

5 Constructing stable applications

Once a system has clearly defined the set of runtime interfaces intended for use by applications[14], and is committed to maintain them in an upward compatible way, all *properly constructed* applications will continue to run without change. This raises the question of how we decide that any given application meets those criteria.

13. Some analog to the SUNWprivate symbol set that *Solaris* system libraries use to indicate unstable inter-library artifact which is not part of the ABI as opposed to stable interfaces that application developers are intended to use.

5.1 `appcert`: Checking applications’ interface use

An immediate benefit of the Solaris ABI is that we can use the definition it provides to decide whether a compiled application (or other software product) uses unstable interfaces. This can be done by a tool which:

- Determines all bindings an application makes to interfaces in Solaris’s libraries.
- Extracts the system’s interface definition information (Public vs. Private interfaces) from the Solaris libraries.
- Warns of any bindings made directly from the application to Private (non-ABI) interfaces in the libraries.

We have written a tool for *Solaris* that performs the above examination and one or two other checks for potential binary instabilities (for more information see [appcert]).

Implemented as a Perl script, `appcert` relies on two important *Solaris* system utilities: To determine an application’s runtime dependencies (both the libraries and specific per-library symbol bindings) `appcert` relies on a feature of the *Solaris* runtime linker (`ldd(1)`)[15]. Next, for each *Solaris* system library the application depends on, `appcert` uses the `pvs(1)` utility to determine the library’s ABI (its Public vs. Private symbols).

Some additional checks related to binary stability are also performed by `appcert`. In particular, the static linking of *Solaris* archive libraries (e.g., `libsocket.a`) are flagged, as well as calls to certain specific interfaces—whether individual symbols or entire libraries, known to have caused binary breakage in earlier releases.

14. This also applies to any other layered software product that is not part of the system (in the sense that it does not such an integral part of the core system software that it must be re-built and reissued as a part of every release of the system software product).

15. ldd is run with the environment variable LD_DEBUG set to “files,bindings”.

6 Benefits

Library versioning, as present in both *Solaris* and *Linux* provides a finer grain solution to the minor-revision rendezvous problem described above. An application which has been constructed using versioned libraries records the name(s) of the version(s) containing the interfaces that it uses, and that it thus requires to be present in a library on a runtime platform. Beyond location of a library matching the major revision level needed by the application on the runtime platform (an exact match of the *so-name* recorded in the application binary), the runtime linker now also ensures that the minor version dependencies recorded in the application are present within the library.

Library scoping has been applied to eliminate a class of library internal interface from external visibility. Dynamic linking of the libraries is sped up by scoping, since scoped symbols are removed from the dynamic symbol table (.dynsym): Since scope-reduced symbols become LOCAL symbols, references to those symbols (within the library) are resolved statically at the time the library is constructed. Dynamic relocations are no longer required for these symbols.

In *Solaris*, library versioning has also been applied to define the ABI—a stable, upward compatibly evolving interface for applications, and to distinguish this from a set of interfaces exposed by libraries which reflect part of the system's internal implementation. This serves as the foundation for ensuring the integrity of successive system releases, and for establishing stability in the installed base of applications and software products that rely on the system.

7 Conclusions

Given that the enabling technology is now present in the GNU linker, and has been demonstrated in its application to `glibc`, it strikes us as highly desirable that additional libraries used by *Linux* developers (e.g. `XFree86` and `libgtk`) adopt versioning practices consistent with those used by the GNU `glibc` libraries. An important part of this will be to identify and advocate a set of *policies* to be used—especially important considering the number of independent developers contributing to *Linux*-based systems. The more libraries that carefully define and manage the evolution of their external interfaces, the smaller is the chance for binary incompatibilities to arise for applications that depend upon them. And the more *uniform* the set of practices for implementing library interface definitions, the more practical will be developers ability to understand and apply that in the software products that they construct.

7.1 A *Linux* ABI

We are convinced that the GNU `glibc` project (and other *Linux*-related library projects) would benefit if a GNU/*Linux* ABI were defined for these libraries. This could be done, just as in *Solaris*, by adding an analog to SUNWprivate (for example, a "GLIBC_PRIVATE" for the GNU `glibc` package), to indicate the system-internal (non-ABI) symbol set. Currently both application interface (ABI) and system-internal interface (non-ABI) symbols appear to be exported together.

If the *Linux* community discovers these practices to be effective, it should be as natural to define all library interfaces as it was for *Solaris*. In fact, due to the more distributed and modular nature of open source development, it may prove even more fruitful to apply these practices. Further, due to the independent development and release of many of the libraries used in *Linux*-based systems, it may be necessary to explore additional classifications beyond the "Public" and "Private" used in *Solaris*, perhaps to identify and version inter-library interfaces (those between separately-released collections of libraries). While it may appear that this use of symbol versioning only applies to monolithic "cathedral" systems like *Solaris*, it should be noted that Sun also applies its versioning scheme to libraries from outside Sun (e.g., the CDE and X11 libraries) released as part of *Solaris*.

Given the similarity of mechanisms, the definition and use of the ABI to cultivate a base of increasingly stable applications, as in *Solaris*, could easily be done in *Linux*. As an initial step, and proof of concept, we have developed a prototype of `appcert` on *Linux*. But while the tool itself is a necessary element of the solution, the identification of a core set of system libraries for *Linux* systems and the definition and stabilization of their interfaces is needed to realize the full value. Such considerations might serve as the basis for a broader discussion of what libraries constitute the core system interface for *Linux*, and what interface definition and versioning practices might be useful to the open source community and development process.

Our ultimate desire is that a set of normalized practices for library interface definition and management of compatibility will be identified that are sufficient for common and widespread use in the industry.

7.2 Compatibility across *Linux* systems

Sun has benefitted from the library versioning practicesdescribed, both by defining *Solaris's* system interfaces and in managing their upward compatible evolution. We are excited to see these mechanisms and similar practices adopted by the GNU `glibc` project, and hope that the practices will be developed and applied more broadly in open source library development.

A significant opportunity that arises from the definition of an ABI and library versioning, is the ability to compare the system interface provided by different system or product releases. While most recently this has been used in *Solaris* to maintain upward compatibility for successive releases of a *single* product, definition of an ABI in the *Linux* environment could serve to enable *cross-product* binary compatibility, so that a software product build on one *Linux*-based system (such as a Caldera release) could be run successfully on others (such as *Linux*-based distributions released by RedHat, Debian, SuSE and so on). This could prove important to avoid a Balkanization of the interface as offered by different *Linux*-based releases, and perhaps critical to the success of the open source efforts related to its ongoing development.

References

[appcert] “Solaris appcert tool”, available at URL: http://www.sun.com/developers/tools/appcert.

[Gingell 87] Robert A. Gingell, Meng Lee, Xuong T. Dang and Mary S. Weeks, “Shared Libraries in SunOS”, USENIX Conference Proceedings, pp. 131-147, Summer 1987, Phoenix, AZ.

[GNU_ld] Info pages for the GNU linker (ld). File:ld.info, Node:VERSION, GNU glibc version 2.x.

[Huiz 97] Gerritt Huizenga, Dynix library versioning practices, personal communication, 1997.

[Johnson 98] Michael K. Johnson and Eric W. Troan, “Linux Application Development”, Addison Wesley Longman Inc., (c) 1998, Reading, MA, ISBN: 0-201-30821-5.

[Solaris LLM] “Solaris Linker and Libraries Manual”, available at URL: http://docs.sun.com/ab2/coll.45.13

[SysVABI] “System V Application Binary Interface”, ISBN 0-13-100439-5, UNIX Press (Prentice Hall), Englewood Cliffs, N.J. (c) 1993.