Why Do Software Packages Conflict?

Cyrille Artho, Kuniyasu Suzaki
Research Center for Information Security
AIST
Umezono 1-1-1, Tsukuba,
Ibaraki 305-8568, Japan
{c.artho,k.suzaki}@aist.go.jp

Roberto Di Cosmo, Ralf Treinen, Stefano Zacchiroli
Univ Paris Diderot, Sorbonne Paris Cité
PPS, UMR 7126, CNRS, F-75205
Paris, France
roberto@dicosmo.org, treinen@pps.jussieu.fr
zack@pps.univ-paris-diderot.fr

***Abstract*—Determining whether two or more packages cannot be installed together is an important issue in the quality assurance process of package-based distributions. Unfortunately, the sheer number of different configurations to test makes this task particularly challenging, and hundreds of such incompatibilities go undetected by the normal testing and distribution process until they are later reported by a user as bugs that we call "conflict defects".**

We performed an extensive case study of conflict defects extracted from the bug tracking systems of Debian and Red Hat. According to our results, conflict defects can be grouped into five main categories. We show that with more detailed package meta-data, about 30 % of all conflict defects could be prevented relatively easily, while another 30 % could be found by targeted testing of packages that share common resources or characteristics. These results allow us to make precise suggestions on how to prevent and detect conflict defects in the future.

I. Introduction

A. Package-based Software Distributions

Modern software distributions are organized into packages. A software package is a self-contained unit that can be installed or removed independently of other packages, as long as dependencies are met. A package manager controls such administrative tasks; compared to unmanaged installations, the benefits of a package-based approach are the ability to automatically install, upgrade, and remove packages without the need to remember installation locations or which files are affected by a change.

In real software, this ideal state is not easy to achieve, due to dependencies between software packages, and interactions between software belonging to different packages. Dependencies arise because some packages provide functionality used by others. Interactions occur on shared resources, such as files, and because packages may provide components that can be combined into a larger system (such as client and server packages communicating together).

Dependencies restrict the ability to freely install, remove, or upgrade packages. If a package a depends on another package b, a package manager automatically requires b to be installed when a is requested to be installed. Furthermore, package b cannot be removed as long as a is still in use. Finally, upgrades of one package often require a simultaneous upgrade of related packages. In addition to this, there is a notion of *conflicting* packages: two packages may use the same resource or provide the same service in a way that is incompatible, so only one of these two packages may reside on a system at any given time.

In package-based software distributions, so-called *package meta-data* describes dependencies and relations between packages. Most Free and Open Source Software (FOSS) systems are managed in that way. Meta-data contains information about dependencies of packages, and conflicts between them. At the time of writing, meta-data covers relations among packages at the package level; dependencies and conflicts are indicated by package, not by the actual resources a package provides or depends on. Different packaging mechanisms have different ways of giving more fine-grained information on packages. *Features* (as used by the Red Hat package manager) may represent individual libraries. So-called *virtual packages* are sometimes used as placeholders for actual resources or services provided by a package.

Unlike fully fledged packages, such placeholders do not include any actual installation of programs. Both mechanisms leverage the package dependency system, representing resource dependencies as package dependencies. However, neither mechanism constitutes a generic way of describing resources, such as files, network ports, or system services, in an accurate and fine-grained way. There is no formal definition linking virtual packages to actual resources (or vice versa); definitions are made by package maintainers instead of automated tools.

An abridged example of the meta-data of the Debian package for the `mutt` mail user agent is given in Figure 1. This package depends on a long list of library packages only partially shown in the figure. It also has a recommendation (that is a weak form of dependency) on a virtual package `mail-transport-agent`. That package in turn is provided by several other packages like for instance `exim4` or `sendmail`, and it provides itself two virtual packages, `imap-client` and `mail-reader`. In addition to that,

```
Package: mutt
Architecture: amd64
Version: 1.5.21-5
Replaces: mutt-utf8
Provides: imap-client, mail-reader
Depends: libc6 (>= 2.3.4),
  libcomerr2 (>= 1.01), ...
Recommends: mail-transport-agent, ...
Conflicts: mutt-utf8
```

Figure 1. Excerpt of Debian meta-data.

it conflicts and replaces a package named mutt-utf8, which was useful in the past but is now obsolete (since superseded by mutt) and which has been removed from the Debian archives. However, one has to assure smooth upgrade from situations where that old package is still installed. This smooth upgrade is achieved through a combination of Conflicts and Replaces as shown in the example.

B. *Conflict Defects*

Conflict defects occur if the combination of multiple packages results in a defect that is absent otherwise. Package meta-data—and in particular explicit conflict declarations—may indicate such defects, which prevents conflicting combinations of packages from being installed. However, conflict defects may still arise in practice. The reasons for such defects are manifold: packages are not just bundles of files, but include pre-installation and post-installation scripts. These scripts are unrestricted, Turing-complete programs running with full system (root/administrator) access. It is impossible in general to capture the full side effects of these scripts with a formal description. Actual conflict defects might simply go unnoticed through a testing phase or might be impossible to describe properly. The same problem arises when executing the software provided by these packages. Therefore, a complete logical analysis of package behavior is not possible. Nonetheless, as this paper shows, steps can be taken towards covering certain types of common conflict defects that are not automatically verifiable with current tools.

Another problem arises from the fact that a significant part of package meta-data is provided manually, by package maintainers. It is therefore a challenge to keep such meta-data up to date and accurate. This challenge becomes especially daunting in the presence of a huge number of software packages in distributions such as Debian, where the number of packages available currently exceeds 30,000 [14].

As a consequence of this, bug reports referring to conflict defects between packages are becoming frequent. This paper investigates the origin of such defects and tries to answer the following questions:

1) What are the main reasons why conflict defects arise?
2) Are there common categories of conflict defects?
3) Can these problems be addressed by using existing tools, or is there a need to improve them, or create new ones?
4) Is package meta-data currently being used, accurate and sufficient? Is there a need to automatically verify such meta-data for accuracy, or is there a need to use additional meta-data for a more accurate notion of package conflicts? In other words, are most or all possible conflict defects covered by meta-data?

This paper is organized as follows: Section II describes related work. Section III shows two case studies on conflict defects in Debian and Red Hat, with a detailed evaluation of different kinds of conflict defects. Section IV discusses the results and proposes possible strategies for remedying problems found, and Section V concludes and outlines future work.

II. Related Work

A. *Software Packaging*

Software packages are a well-known example of the component models that have originated from the field of component-based software engineering (CBSE) [19], [3]. Packages fit within common component definitions, but the raise in their popularity—started with the advent of FOSS package managers such as the FreeBSD porting system [17], APT [10], Yum, etc.—has highlighted very specific challenges related to their deployment [6]. Some of those challenges are being addressed relying on package meta-data and their formalization.

Seminal work [9] has shown how to encode the installability problem for software packages as a SAT problem, established the (NP-Hard) complexity of the problem, and shown applications of the encoding to improve the quality of package repositories by avoiding non-installable packages. Based on the same formalization, various quality metrics have been established, such as strong dependency and sensitivity [1] (to evaluate the “importance” of a package in a given repository) and strong conflicts [5] (to pinpoint packages which might hinder the installation of several other packages). In the same vein, package meta-data has also been used to predict future (non-)installability of software packages [2]. The abundance of studies that rely on package meta-data testifies the importance of the correctness of meta-data.

On the other hand, studies on package meta-data correctness like this one, seem to be scarce. At the same time, a few testing tools can be found in the realm of Quality Assurance (QA) of FOSS distributions to discover *symptoms* that might then lead, a human, to discover errors in package meta-data. To name one, the “file overwrite” [20] initiative helps in discovering undeclared conflicts among packages in the Debian distribution.

B. Alternatives to Globally Managed Software Packaging

As an alternative to globally managed software packages that are organized in a fine-grained hierarchy, self-contained packages including all sub-components, sometimes called *bundles*, are sometimes used. Such bundles include the application and all libraries it depends on, linked statically [12]. This contrasts to FOSS distributions where libraries are shared, and generally required to be shipped as separate packages—see for instance [8], "convenience copies of code"—in order to ease the deployment of (security) upgrades. In a system using bundled software, all applications using the library in question need to be updated separately. This usually entails a longer period during which a system is vulnerable, because some software bundles may be provided by third parties.

An advantage of self-contained software bundles is the ease of testing and deployment, as system-specific configurations and libraries have only limited impact on the software bundle. However, statically linking all libraries used by a bundle requires much disk space. If many applications include the same statically-linked libraries, these libraries are duplicated within the same system. Deduplication addresses this problem [4], [18]. Memory and storage deduplication merge same-contents chunks on block level, and reduce the consumption of physical memory. By sharing identical chunks of storage, logical-level redundancies caused by static linking are resolved on the physical level.

III. Evaluation of Conflict Defects

A. Repositories Used in the Case Study

The evaluation of existing conflict defects was carried out on two publicly accessible bug repositories: The Debian bug repository [13] and Red Hat's bugzilla [16]. These represent the two of the most widely used FOSS distributions for the past 10 years. Red Hat's repository also contains bugs related to Fedora, a community distribution on which Red Hat Enterprise Linux is based.

To get a summary of the Debian bug repository, a snapshot of the Ultimate Debian Database (UDD) [11] was taken. This database contains key data of all *open* bugs at that time, such as bug ID, title, and the affected package. The snapshot, taken on January 23rd 2011, contains 79,936 bugs.

For Red Hat, no such summary snapshot is available; however, bugzilla offers a web-based search that returns all data in XML format. Like in the Debian case study, the search returns matches on all open bugs. The searches on Red Hat's database were carried out on February 4th, 2011. While the exact total number of open bugs at that time is not known (because a search with no filter is not possible), the highest number (bug ID) returned by the search, roughly matches Debian's; furthermore, the number of search results is also comparable. This leads us to believe that the samples in both case studies are taken from repositories of comparable size.

Table I
NUMBER OF MATCHES PER KEYWORD IN DEBIAN BUG DATABASE.

Keyword	Matches	Refined matches
break	575	161
conflict	252	85
overwrite	102	44
total	929	290

Table II
NUMBER OF MATCHES PER KEYWORD IN RED HAT BUG DATABASE.

Keyword	Matches	Refined matches
break	166	111
conflict	119	106
overwrite	19	9
total	304	226

B. Methodology

1) Automated Search: As the bug database is too large to be analyzed manually, the selection of bugs is first narrowed down by a keyword search. We chose three keywords to search for: "break", "conflict", "overwrite". The first two words are generic descriptions of conflict defects and often appear in the form "a breaks b" or "a conflicts with "b". The last keyword describes one of the most common inter-package problems, where one package overwrite a resource needed by another package.

Tables I and II give an overview of all the matches in the search. A total of 929 bugs match the initial search on the Debian repository, and 304 bugs match on Red Hat's bugzilla. Some of the matches contain more than one keyword and are therefore duplicates. Our aim is not to get an exact number of how many conflict defects there are in total. Rather, we want to know what types of conflicts occur more often than others, relative to the total number.

We then narrow the search to eliminate bug reports that describe problems that relate to one package alone, rather than a conflict between two packages. For example, "overwrite" could appear in a bug report related to overwriting text in a text editor. Indeed, an initial manual evaluation on Debian shows that about half of all bug reports found in the initial search are not related to conflict defects. To make the results more accurate, the search is refined to include only bug reports out of the initial selection, where the title contains the name of another package. This may filter out more bug reports than necessary (decreasing recall, in search terms), but makes the results much more precise. To avoid excluding too many packages, (version) numbers of packages are not included in this filter, even if the package name itself contains a version number. A manual check shows that this filter is good approximation of a manual selection of true conflict defects.

As shown in Table I, the refined selection on Debian contains 290 matches. Some of these matches contain multiple keywords in the title; 241 of them are distinct bug

Table III
BUGS EVALUATED IN DETAIL.

	Debian	Red Hat
Bugs after initial search	929	304
Having package name in title	290	226
Manual filtering of title contents	190	226
Bugs that are not conflict defects	51	43
Actual conflict defects	139	183

reports. On Red Hat, all 226 refined matches are distinct bug reports. On Debian, further manual post-processing of that list removes another 51 items, where the title indicates clearly that those are not conflict defects. This leaves 190 bug reports where, judging from the title of the report, a possible conflict defect is reported.

At this early stage, checking the bug description filters out a much smaller number of bugs on the Red Hat case study. We think that this is partly because more professional developers and proportionally fewer volunteers contribute to Red Hat's bug database. This may lead to the language on Red Hat's database being more uniform, making a keyword search more precise. Another reason is that a particular category of bugs, a conflict between 32-bit and 64-bit packages (see below), occurs often in Red Hat; this improves search precision. The second stage of the the evaluation on Red Hat's bugzilla is performed on the remaining 226 bugs.

Table III summarizes the search and selection stages: An initial keyword search yields a large number of bugs; these matches are refined by keeping bug reports where the title includes a package name, hinting at the existence of a conflict defect. In the case of Debian, the number of bugs is further reduced by a manual analysis of the bug title. For Red Hat, the full bug text is analyzed in all remaining cases, because the remaining number was smaller after accounting for conflicts between 32-bit and 64-bit packages.

2) Manual Evaluation: The final stage of analysis is done manually, requiring the full information on each bug. In the initial web-based searches, these detailed results are not returned. Both the Debian summary database (UDD) and Red Hat's search return only summary data. The bug IDs returned in the summary link it to the detailed bug description.

The actual bug reports are obtained by downloading them from the web page representing the corresponding bug repository [13], [16]. Manual study and categorization of the bugs rules out a number of possible candidates as being problems related to a single package rather than a combination of packages, as shown in Table III. Bugs that are not counted include the following:

- bugs that are clearly not reproducible,
- bugs of which the description is unclear,
- bug reports which are later retracted as incorrect, and,
- in Red Hat, two bugs where access to details is denied to the public.

This leaves 139 and 183 genuine conflict defects, respectively. A subset of these bug reports is evaluated in a first sample, to come up with a categorization of bug reports that would not be too coarse (giving only a few rough classes of bugs) and not be overly fine-grained either (putting most bugs into a category of their own). After that, all bug reports are classified according to these criteria, or eliminated as not being conflict defects. The categorization is refined during the process, to merge similar categories where one category has few elements. This is similar to a clustering algorithm, except that the measure of similarity between categories is subjective, as the semantics of natural language cannot be easily quantified with today's technology.

3) Possible Sources of Bias: Our study was designed without any personal bias towards existing software distributions or packages. The packages in question mentioned in the bug reports are not developed by us. Nonetheless, our study contains sources of possible bias.

Our first step is based on filtering the title (or summary) of each bug report against given keywords and package names. We are aware that this initial filter may be too strict in some cases, and filter out some reports that indeed pertain to conflict defects. As mentioned earlier, though, the objective of our study is to know the relative characteristics of bug reports concerning conflict defects, so the overall prevalence of conflict defects within all bug reports is not the focus.

Our second step is a manual evaluation, which is by definition imperfect because it is done by a human. We have made our best efforts to classify the data consistently into distinct categories, but we are aware that these categories are not formally defined and therefore not completely unambiguous. However, as shown in Sections III-D and IV, the overall trends found by our study are quite clear, and do not depend on each single classification being accurate.

C. Repository Characteristics

With respect to the recentness and lifetimes of bug reports, the repositories are similar but also show interesting differences. Figure 2 shows a histogram of the frequency of bug reports per year, for the final 190 and 226 cases.[1] The number of bugs is shown by the year in which they were submitted, and the year in which they were last modified. This information is taken from the detailed description, and it is not directly available for the entire repository. However, we think that our sample illustrates an overall trend.

Both repositories contain open bug reports going back several years, with most of the bug reports being very recent (from the last two years). Debian has a markedly higher number of bug reports going back more than a few years, while older bugs are almost absent in Red Hat's repository. Furthermore, all bug reports in Red Hat's database are

[1] The choice of this sample arose from the need to download the detailed bug reports for these cases.

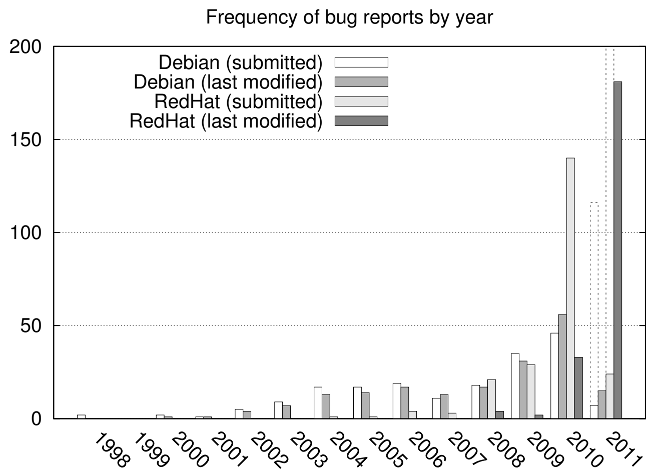


Figure 2. Characteristics of both bug repositories.

modified frequently, and most of them have been modified in the last 12 months.

For the year of the case study itself (2011), the dotted box in Figure 2 shows the projected number of bugs in that year, based on an extrapolation of the number of bugs during the days of 2011 before the snapshot was taken. This estimate, 116 and 257 bugs, respectively, shows that the exponential growth of open bugs towards recent years continues. This is due to a "half-life" of bug reports, which indicates a probability for any bug to be closed at a given time. For the time of the last update, such an extrapolation cannot be done well, because updates of older bugs cause the timestamps of these bugs to move within the histogram.

This overview may suggest that Red Hat frequently fixes old bugs, or at least updates them. It turns out that the latter is indeed the case, via automated updates of bugs concerning packages that are no longer supported. However, it does not seem to be the case that Red Hat fixes old bugs at a higher rate than Debian. Rather, old bug reports are often obsoleted: If a bug report relates to software that is no longer in today's Red Hat distributions, they are first updated with an end-of-life warning, and later closed automatically. This process contains a standardized message and is probably at least partially automated.

Debian has no practice of automatically closing bug reports related to outdated or obsolete packages, with the notable exception of bugs belonging to packages that get removed from the Debian archive.

D. Categorization of Conflict Defects

As described above and shown by Table III, 190 (Debian) and 226 (Red Hat) bug candidates are subject to manual classification. The manual evaluation categorizes bugs into a hierarchy of categories. The categories for both repositories are identical, except for one specific type of bug that does not occur in Debian.

On Red Hat's bug repository, a large number of bug reports refers to conflicts between 32-bit and 64-bit versions of the same package. These packages can be installed in parallel but doing so may lead to a corrupt system, as described below. These cases can be counted by matching the bug description against one of the following keywords/phrases: "multiarch conflict", "multilib conflict", or "i386/x64". 57 bugs on Red Hat's side fit into this category.

Several CPUs are nowadays able to run programs that use different register and pointer sizes; a common example is a 64-bit capable x86 CPU that can also run 32-bit executables in *legacy mode*. To properly run an executable in legacy mode, all the shared libraries it needs must also be available as 32-bit libraries. Distributions have therefore deployed support to install—side by side and under different paths, to avoid file conflicts—32-bit and 64-bit versions of the same library packages on the same machine.

As it happens, not all files that from a library package are objects that need to be differentiated according to their mode; for instance, documentation files and other architecture-independent data can be shared across different modes of the same library package. To resolve conflicts on files that are common to both versions, a possible solution is to move these files into a separate architecture-independent package depended upon by architecture-dependent libraries. This is the solution chosen by Red Hat-based distributions, and it requires adapting all library packages across the whole distribution. An alternative solution (chosen by Debian-based distributions) is to amend the packaging system to allow sharing of identical files across different modes of the same package.

The different solutions chosen and the different state in adoption of multi-arch explain the differences among the occurrences of multi-arch bugs in Red Hat and Debian. As this is a transient adaptation phase, we elide this category for the remainder of this section.

The remaining bugs are classified into five categories:

1) Unavailability or Inaccessibility of Shared Resources: Shared resources are often files, but also include other unique system resources such as network ports or C library function names. Whenever a conflict occurs directly on a file, the conflict is caught at installation time by the package manager (see Figure 3 for an example). This handling is safe, but unsatisfactory: if a list of files used were provided beforehand, then an enhanced package manager could prevent an installation attempt that is bound to fail. On the other hand, other conflicts, such as name clashes in libraries, may not be detected until an application is used at run-time.

To summarize, bugs in this category are caused by the *unavailability* or *inaccessibility* of shared resources (e.g. due to mutual exclusion of the resource and ownership by "others").

2) Conflicts on Shared Data, Configuration Information, or the Information Flow Between Programs: Configuration

```
Unpacking gcc-avr (from .../gcc-avr_1%3a4.3.0-1_amd64.deb) ...
dpkg: error processing /var/cache/apt/archives/gcc-avr_1%3a4.3.0-1_amd64.deb
 (--unpack):
 trying to overwrite '/usr/lib64/libiberty.a', which is also in package binutils
dpkg-deb: subprocess paste killed by signal (Broken pipe)
Errors were encountered while processing:
 /var/cache/apt/archives/gcc-avr_1%3a4.3.0-1_amd64.deb
E: Sub-process /usr/bin/dpkg returned an error code (1)
```

Figure 3. File conflict found when trying to install a package.

```
Red Hat Bugzilla - Bug 593402
Package: Spacewalk
Version: 1.0
Severity: medium
```

Cobbler-web breaks /rpc/api over http (which breaks taskomatic)
This is caused by the fact that both the cobbler_web.conf and zz-spacewalk-server.conf contain a `<VirtualHost>` section. The cobbler_web.conf specifies one for port 80 (`<VirtualHost *:80>` while the zz-spacewalk-server.conf defines one for the default (it's just `<VirtualHost>`).
As the VirtualHost for port 80 (as defined in the cobbler_web.conf) has preference over the default VirtualHost (as defined in zz-spacewalk-server.conf) the rewrite engine doesn't get enabled for port 80. This results in breakage of the spacewalk rewrite rules which are mentioned in zz-spacewalk-www.conf.

Figure 4. Report on conflicting configuration data.

```
Red Hat Bugzilla - Bug 606243
Package: canorus
Version: 0.7.6
Severity: medium
```

Installation of canorus breaks operation of prelink
Description of problem:
If canorus is installed prelink fails.
I get daily cron mails with the following content:

```
/etc/cron.daily/prelink:
/etc/cron.daily/prelink: line 47: 32734
Aborted   /usr/sbin/prelink -av $PRELINK_OPTS
>> /var/log/prelink/prelink.log 2>&1
```

Figure 5. Report of a conflict arising from the interaction between packages.

information is often found in `/etc`, while shared data may be located elsewhere. Information flow refers to function calls or communication via pipes or a network. There are two basic cases where conflicts occur on data or communication:

1) An installation action of a package changes the configuration such that either the syntax of a configuration file is broken (made unreadable for the parser used by another tool), or its semantics changes in an incompatible way with respect to previous expectations.
2) A change in the data format between versions of an application, which requires updating other components; the lack of an appropriate newer version of other components, or the lack of a declaration of such, causes a conflict.

In both cases 1 and 2, the conflict usually only becomes evident at run-time. Sometimes, the problem can be avoided by having an installation script leave a configuration file unchanged if it has been modified by a user. Figure 4 shows a typical case of a semantic conflict in configuration entries generated by two different packages.

Bugs in this category are caused by *incorrect data* in shared resources or interfaces.

3) Interactions between Packages: In some cases, a package a using another package b makes a previously undetected fault in b evident; it is possible that other use cases for b could produce the same problem, so the failure can (at least in theory) be reproduced using b alone. In other cases, the combination of a and b is necessary for those packages to fail, and either package would work fine without the conflicting package being present (see Figure 5).

The bugs have in common that they are observed as a conflict arising from the *interaction between packages.*

4) Package Evolution Issues: When a software distribution evolves, packages may be renamed or split up into multiple packages, or several packages may be merged into one. This may require updating meta-data in other packages for the distribution to remain consistent. Furthermore, version changes with a package may also require meta-data changes due to possible incompatibilities mentioned above. Unfortunately, meta-data changes are not automated, and are primarily the responsibility of the maintainer of a given package. This causes a potential for meta-data to be outdated and not reflect a correct state anymore. The bug shown in Figure 6, for example, was due to an attempt to implement a transition from package `ttf-telugu-fonts` to `fonts-telu`; in this case, the maintainer used incorrect package relations.

Problems in this category arise due to incorrect or outdated *meta-data.*

5) Spurious Conflicts: The last category represents cases where two packages are incorrectly classified as conflicting, although there is no conflict, at least not for the current version of these packages. An example report of a spurious conflict is shown in Figure 7.

Table IV
OVERVIEW OF ALL CONFLICT DEFECTS FOUND IN THE TWO BUG DATABASES.

Number of conflicts		Conflict type
Debian	Red Hat	
0	57	32-bit/64-bit binary conflict
43	38	access to/names of files and similar shared resources
22	22	package provides same file as other package
8	6	package (de-)installers modifies file or file permission, or deletes file used by other package
3	5	file/directory name conflict (for names including version number etc.)
10	5	clashing library symbols/function names/device names
48	34	file/API/data/configuration format/resource management
20	12	update/installation script breaks configuration, file format, or resource management
14	8	package breaks on uncommon or user-defined configuration/setting
4	6	package use (post-install) overwrites/breaks configuration files
10	8	API/file format change between different package version breaks other package
21	41	rare (previously untested) combination of packages
13	22	defect in one package made visible by failure of other package/functionality
8	19	uncommon combination of packages makes one or more packages always fail
19	12	package evolution (split/merge/change) or faulty meta-data results in conflict
10	6	incorrect/outdated dependency meta-data (requires/conflicts)
9	6	package renaming/split/merge results in incorrect meta-data of other package
8	1	spurious conflict declaration prevents compatible packages from being used
139	183 (126)	total (in parentheses: total excluding 32/64-bit binary conflicts)

```
Debian bug number: #662988
Package: ttf-telugu-fonts
Version: 2:1.0
Severity: serious
```

Hi, ttf-telugu-fonts is not installable in sid: (...)
The problem is that ttf-telugu-fonts depends on fonts-telu which in turn breaks: ttf-telugu-fonts. This probably should be ttf-telugu-fonts (<<2:1.0).

Figure 6. Report of incorrect meta-data.

```
Debian bug number: #559161
Package: libopenmpi-dev
Version: 1.3.3-2
Severity: serious
```

The libopal-dev and libopenmpi-dev packages were marked as conflicting to resolve bug #404003; the problem was file ’/usr/lib/libopal.so’ contained in both packages.
Since at least lenny this library was renamed to ’/usr/lib/libopen-pal.so’ in libopenmpi-dev package, so the conflict does not exist any longer.
There are no other conflicting files in these packages, so the conflicts tag should be removed.

Figure 7. Report of a spurious conflict.

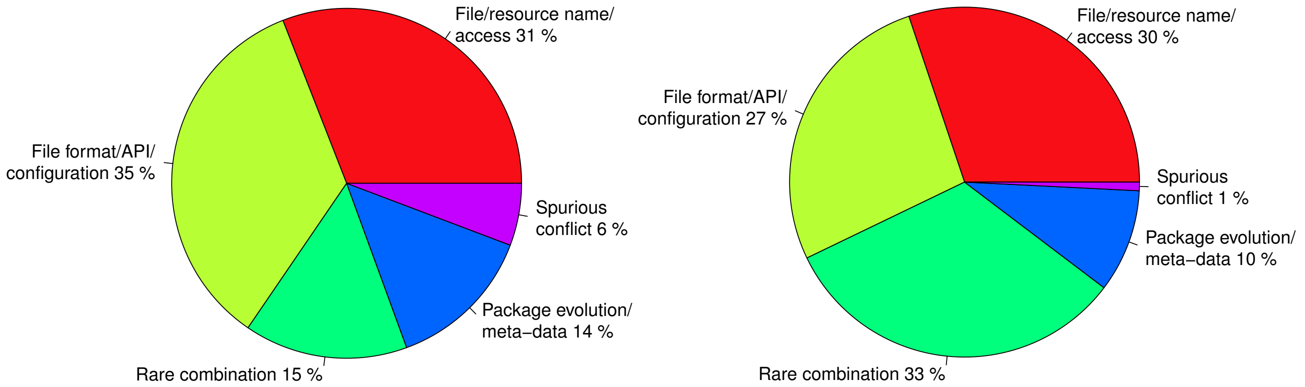
E. Summary

Table IV and Figure 8 show an overview of the classification into these five categories. Larger categories are split up into smaller groups to get a more detailed picture. Conflicts between binaries for different architectures (on Red Hat) are excluded in Figure 8b. While human error in individual classifications is possible, the results are overall quite clear for larger categories. Some trends are evident:

1) Resource conflicts represent about 30 % of all conflicts (43 and 38 cases in total). About half of these conflicts are on files and caught by the package manager at installation time; other similar conflicts may not be caught until a package is actually used.

2) Conflicts on configuration, and to a lesser degree, the format of shared data, are equally common. In many cases, syntactic problems cause a conflict between packages; the most common reason is the automatic modification of configuration files by installation scripts (20 cases in Debian, 12 in Red Hat). These installation scripts are likely tested for common configurations, but may not behave as expected for less common settings. While syntactic problems are prevalent, unintended semantic changes are also a significant problem, both during and after installation. This is compounded by the fact that many configuration files have to be customized by the user before a package can be used. When editing a file, the formatting of a configuration file may see subtle changes that are correctly dealt with by the packaged software itself, but not by the installation scripts that manage the package.

3) Other problems between packages that are usually not installed together represent another significant share. The huge number of available packages makes it impossible to test all combinations (or even just all pairwise possible combinations) of packages together, so a conflict often goes undetected until reported by a user. In Red Hat, the number is fairly large because many problems are reported for specific laptop hardware configurations where kernel modules (driver packages) did not behave well. It seems that the use of Debian in such cases is less common, accounting for a lower percentage of such bug reports.



(a) Debian

(b) Red Hat

Figure 8. Categorization of conflict defects in our case study.

4) Conflicts on meta-data level, often caused by package evolution, contribute about 10 %.
5) Incorrect (or outdated) information on conflicting packages sometimes occurs as well, which does not create a conflict defect per se, but instead prevents two packages from being used together even if this is possible in principle.

IV. Discussion

A. Addressing Each Type of Conflict Defect

The previous section has given a categorization of conflict defects based on empirical data. We now propose possible solutions that can potentially cover some or all instances of each class of conflicts.

1) Conflicts on Files and Similar Shared Resources: Conflicts on shared resources are not directly covered by existing meta-data, although they may be implied by package-level conflicts. Work is in progress to systematically test package installations against overwriting files provided by another package [20], at least in Debian. As an alternative to this, *file diversions* enable a package to install files at a different location; work is in progress to automate this.[2]

This case study shows that while the majority of such conflicts occurs at file level, file *permissions* (and ownership) rather than just file names, and possible file/directory renaming actions during package upgrades, should also be considered. Finally, coverage of similar resources such as network ports and function or library names would further augment the ability of such tools to detect conflicts proactively.

More detailed meta-data will require much more space than existing (rather compact) package meta-data. We propose that some extra meta-data is generated and used only by developers and package maintainers. As it covers possible conflicts proactively, at development time, not all fine-grained meta-data needs be included in the final distribution. We think that most or all of such resource-related meta-data can be extracted automatically by static analysis or run-time analysis. Automation would eliminate extra effort from package maintainers.

This proposition distinguishes itself from existing mechanisms such as features or virtual packages in that extra meta-data is directly linked to underlying resources, in a formally defined way. This makes it possible to generate such meta-data automatically if dependencies on the underlying resources are known.

2) Conflicts on Shared Data: Conflicts on configuration files, file formats and API versions are also common, and clearly demonstrate the need for systematic testing against such conflicts. In the light of testing against overwriting files [20], inter-package tests should also be automatically run against conflicts on shared data. This is much more difficult to automate, and only feasible for packages that include automated regression tests.

The problem is that regression tests are primarily used by developers, and less often by package maintainers, not to mention end users. Because of this, combined with the fact that a unit test failure does not automatically imply that a package is unusable, regression tests are currently not covered by package meta-data. This makes them inaccessible to today’s package management tools, and precludes the automated discovery of such intricate conflicts. However, at a lower level, many source-level distributions have a “make test” or “make check” build target that automatically performs such tests. In the future, such information could be provided in package meta-data, for package maintainers. Furthermore, on a basic level, certain problems may be found just by executing a program and checking whether its return value indicates an error, or by attempting to start and stop a system service cleanly.

[2] http://wiki.debian.org/SummerOfCode2011/DeclarativeDiversions, retrieved June 2011

3) Interactions between Uncommon Packages: The fact that rare combinations of packages may cause problems is not surprising, given the large number of available packages. Exhaustive testing of package combinations is not feasible, but heuristic-based testing of sets of packages may be. A possible approach may be to install larger subsets of packages, and to narrow down the set of conflicting packages by a systematic search such as delta debugging [21].

4) Package Evolution: Package evolution often brings with it an invalidation of package meta-data. About one tenth of conflict defects in our study is caused directly due to invalid meta-data after larger package modifications (such as splitting a package into two packages). This shows that meta-data needs to be verified for consistency and accuracy. Especially when given a situation with "known good" meta-data (before the modification), automatic verification of the new meta-data is feasible if packages can be tested automatically.

As with other issues described above, meta-data does not cover the requirements of packages in enough detail. For example, take a package a that is split up into a' and a'', because some parts of a are not used by many packages. If a package b depends on a in the old configuration, it is possible that b depends on a', a'', or both packages, in the new configuration. If some of the resources provided by these packages are loaded dynamically by b (at run-time), then verification of the actual software is required to determine the correct new dependency.

5) Spurious Conflicts: Spurious (or outdated) declarations of conflict defects can be responded to, by automated testing of packages that supposedly conflict. This would detect cases where a conflict is resolved in a newer version of a package.

B. Summary

To summarize, we think that a large number of common conflict defects can be discovered more effectively through the following means:

- Identification of potentially conflicting packages through analysis of existing meta-data or package behavior. Such an analysis yields candidates for automated testing, covering bug categories 1–3. We expect that such testing may partially use recent virtualization technologies (e.g. [15], among many others). Virtualization technology may provide both a "sandbox" for executing tests and automated inspection of test executions, to determine the usage of shared resources such as files or network ports. As of recently, distributions seem indeed be interested in proceeding along this direction [7].
- More detailed and accurate meta-data, generated or verified by automated tools. This primarily covers bugs related to the availability of shared resources, and the correctness of meta-data itself (categories 1, 4, and 5).

Extended meta-data should cover files including file meta-data in particular, and as a next step, other system resources such as network ports, shared (global) configuration data, and communication between components. Another aspect currently omitted in meta-data is information about regression tests that already exist in many packages, but are inaccessible on a package level because they are not declared or available in a uniform way. An enhanced set of meta-data for testers and distribution maintainers could cover such testing-related information.

V. Conclusions and Future Work

Conflicts between software packages occur due to a variety of reasons. Conflict defects on shared resources and configuration files are particularly common. The underlying problem is that package behavior at installation, use, and de-installation time is unrestricted, so a complete formal description of package behavior cannot be achieved. However, steps can be taken towards increasing the expressiveness and accuracy of package meta-data, by adding meta-data that is intended for package developers and maintainers.

In our case study, we categorize a large number of conflict defects, and propose possible solutions to common categories of conflicts. Our study uses two snapshots of bugs between packages reported in Debian GNU/Linux and Red Hat Linux (including derivatives such as Fedora). We found that on a broad level, over 80 % of all conflict defects are made up by three categories: conflicts on resource access, conflicts on (configuration or application) data, and interactions between uncommon combinations of packages. Future work includes studying the evolution of packages, and bugs reported, over time by investigating multiple snapshots taken over time.

As a conclusion from our case study, we found that ongoing and future projects can reduce conflict defects most efficiently by (a) identifying and testing combinations of packages that may conflict, (b) generating and using extra meta-data, and (c) checking the validity of (manually provided) meta-data.

References

[1] Pietro Abate, Jaap Boender, Roberto Di Cosmo & Stefano Zacchiroli (2009): *Strong Dependencies between Software Components*. In: *ESEM 2009*, IEEE, pp. 89–99.

[2] Pietro Abate & Roberto Di Cosmo (2011): *Predicting upgrade failures using dependency analysis*. In: *27th International Conference on Data Engineering*, IEEE, pp. 145–150.

[3] Alan W. Brown & Kurt C. Wallnau (1998): *The Current State of CBSE*. *IEEE Software* 15, pp. 37–46.

[4] Christian Collberg, John H. Hartman, Sridivya Babu & Sharath K. Udupa (2005): *Slinky: Static linking reloaded*. In: *Proc. USENIX 2005 Annual Technical Conference*, USENIX, Anaheim, USA, pp. 309–322.

[5] Roberto Di Cosmo & Jaap Boender (2010): *Using strong conflicts to detect quality issues in component-based complex systems*. In: *3rd India software engineering conference*, ISEC 2010, ACM, pp. 163–172.

[6] Roberto Di Cosmo, Paulo Trezentos & Stefano Zacchiroli (2008): *Package Upgrades in FOSS Distributions: Details and Challenges*. In: *International Workshop on Hot Topics in Software Upgrades*, HotSWUp '08, ACM, New York, NY, USA, pp. 7:1–7:5.

[7] Ian Jackson, Iustin Pop & Stefano Zacchiroli: *autopkgtest - automatic as-installed package testing*. Debian Enhancement Proposal 8: http://dep.debian.net/deps/dep8/.

[8] Ian Jackson & Christian Schwarz (2008): *Debian Policy Manual*. http://www.debian.org/doc/debian-policy/.

[9] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jérôme Vouillon, Berke Durak, Xavier Leroy & Ralf Treinen (2006): *Managing the Complexity of Large Free and Open Source Package-Based Software Distributions*. In: *ASE 2006*, IEEE, pp. 199–208.

[10] Gustavo Noronha Silva (2008): *APT HOWTO*. http://www.debian.org/doc/manuals/apt-howto/.

[11] Lucas Nussbaum & Stefano Zacchiroli (2010): *The Ultimate Debian Database: Consolidating Bazaar Metadata for Quality Assurance and Data Mining*. In: *7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, Cape Town, South Africa.

[12] Leon Presser & John R. White (1972): *Linkers and loaders*. *Computing Surveys (CSUR)* 4(3), pp. 149–167.

[13] The Debian Project: *Debian bug tracking system*. http://debian.org/Bugs/. Retrieved March 2012.

[14] The Debian Project: *Software packages in [Debian] "sid"*. http://packages.debian.org/sid/allpackages. Retrieved June 2011.

[15] Red Hat, Inc.: *KVM*. http://www.linux-kvm.org.

[16] Red Hat, Inc.: *Red Hat Bugzilla Main Page*. http://bugzilla.redhat.com. Retrieved March 2012.

[17] Murray Stokely (2004): *The FreeBSD Handbook*, 3 edition. FreeBSD Mall.

[18] Kuniyasu Suzaki, Toshiki Yagi, Kengo Iijima, Nguyen Anh Quynh, Cyrille Artho & Yoshihito Watanebe (2010): *Moving from Logical Sharing of Guest OS to Physical Sharing of Deduplication on Virtual Machine*. In: *Proc. 5th USENIX Workshop on Hot Topics in Security (HotSec 2010)*, USENIX, Washington D.C., USA.

[19] Clemens Szyperski (1998): *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley.

[20] Ralf Treinen (2011): *EDOS-Debcheck: File Overwrite Errors*. http://edos.debian.net/file-overwrites/. Retrieved June 2011.

[21] Andreas Zeller & Ralf Hildebrandt (2002): *Simplifying and Isolating Failure-Inducing Input*. *Software Engineering* 28(2), pp. 183–200.