Типизация ABI

Алексей Турбин

21 сентября 2016 г.

Аннотация

Имеющаяся реализация бинарных зависимостей на основе set-версий позволяет контролировать наличие нужных символов в библиотеках. Показано, что реализация может быть расширена таким образом, чтобы учитывался и тип символа, а не только его имя; так чтобы обеспечить не только наличие нужных функций, но и совместимость их прототипов.

1 Введение

В предыдущей работе [1] показано, что понятие *обратной совместимости* часто является обманчивым; для достижения реальной *бинарной совместимости* между библиотеками и программами (ABI) нужно контролировать не только и не столько придуманные версии библиотек, сколько *разрешимость символов* (т. е. фактическое наличие нужных функций в библиотеках). Там же обсуждается, насколько компактным может быть вероятностное представление множеств, при котором каждый символ представлен n-битным хешем. Вскоре был найден и способ кодирования: *код Голомба–Райса* обеспечивает оптимальное сжатие дельт (расстояний) между соседними хешами [2]. Таким образом, у библиотек в репозитории появились зависимости вида `Provides: libfoo.so.1 = set:...`, а у программ — `Reqruies: libfoo.so.1 >= set:...` (где вместо троеточия идет закодированная последовательность хешей). При сравнении таких *set-версий* rpm выполняет проверку $R \subseteq P$.

После этого было реализовано автоматическое создание `debuginfo`-пакетов, а пакеты в репозитории стали компилироваться с флагом `-g` — с отладочной информацией в формате DWARF. Там содержится описание переменных и функций, в том числе аргументов функций и их типов. Используя эту информацию, символы можно наделить *типом*; тогда при сравнении множеств будет проверяться не только разрешимость символов по имени, но и совпадение их типов (для функций — прототипов). Это напоминает декорирование имен (name mangling) в C++, где к имени функции добавляется информация об аргументах. В обоих случаях типизация является *атомарной*: возможно только полное совпадение; никакой дальнейшей структуры в типизированных символах не усматривается. Если символ меняет свой тип, хотя бы и условно совместимым образом, то образуется другой, новый символ (с другим хешем).

Для языка Си полный учет типов порождает довольно жесткую конструкцию. Если тип изменяется условно совместимым образом, то желательно, чтобы символ

остался прежним. Далее обсуждается, какой может быть *облегченная* типизация. Показано, что для ее реализации сборка с флагом `-g` требуется только для библиотек. Однако сборка приложений с флагом `-g` открывает интересные возможности для *дополнительной* типизации.

2 Типизация

Допустим, функция принимает аргумент типа `int`; а в новой версии тип аргумента меняется на `long`. При этом функция сохраняет совместимость: ведь на 32-битных архитектурах `int` и `long` — это один и тот же тип. На 64-битных архитектурах это два разных типа (32 и 64-битный), но при передаче параметра фактически используется 64-битный тип — либо за счет передачи через регистр, либо за счет выравнивания на стеке [3, с. 23]. Аналогично, при добавлении или удалении спецификатора `unsigned` шансы нормального вызова остаются высокими. Поэтому любой целочисленный аргумент наделяется типом `i`.

Рассмотрим теперь квалификатор `const` вместе с аргументом типа `char *` (Си-строка). При добавлении квалификатора совместимость сохраняется полностью, а при удалении совместимость ухудшается. Мы сейчас выполнили *структурное сравнение* на совместимость, которое противоречит принципу атомарности. Ничего не остается, как только полностью исключить квалификатор `const` из рассмотрения, признав его несущественным. Но можно привести и другие аргументы в поддержку этого решения. Использование `const` часто вызывает затруднения (кроме простейших случаев), см. напр. о типе `argv` в `getopt(3)`. В других языках вместо `const` используется явное задание in/out параметров. В третьих языках, таких как Rust, отслеживается владение (ownership) объектами. Нам не хотелось бы признавать истинной ту или иную языковую абстракцию и проверять бинарную совместимость в соответствии с ее догмами.

Рассмотрим типизацию структур, передающихся по значению. У структуры часто бывает два имени: тег структуры и короткое имя, заданное через `typedef`. Но с точки зрения бинарной совместимости вообще не следует привязываться к имени. Поэтому структуры наделяются типом `bN`, где `N` — размер структуры (`sizeof`).

Массивы типизируются как `aT`, где `T` — тип элемента массива (это касается только глобальных переменных; в параметрах массив преобразуется в указатель).

Указатели тоже хотелось бы типизировать как `pT`; однако, в отличие от массивов, тип `T` может быть непрозрачным (opaque). Кроме того, тип может быть полностью скрыт указателем `void *`. Получается, почти все указатели на данные должны иметь тип `p`; можно лишь учитывать уровень косвенности (тогда, например, аргумент `char *argv[]` будет иметь тип `pp`). Кроме того, указатели на функции можно типизировать полностью. В качестве примера рассмотрим функцию `signal(2)`:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Соответствующий ей символ наделяется типом так:

```
signal (i, p(i)) -> p(i)
```

3 Сопоставление

Как и в имеющейся реализации, для сопоставления требуемых символов с предоставляемыми используется загрузчик `ld.so` — он запускается в специальном отладочном режиме, похожем на `ldd(1)`. Каждый требуемый символ наделяется типом соответствующего предоставляемого символа; подразумевается, что на стадии сборки типы символов соответствуют (по этой причине не требуется сборки клиентского кода с отладочной информацией).

Однако такое сопоставление нельзя считать слабым местом всей конструкции. Сборка пакетов является моментом *наибольшего согласования* между клиентским и библиотечным кодом; при наличии несовместимости код не соберется, либо не отработает `make check`. Можно даже сказать, что одна из главных задач системы управления пакетами — это *перенос гарантий*, достигнутых во время сборки, на другие конфигурации. А именно, с помощью бинарных зависимостей мы пытаемся гарантировать, что при запуске программы символы будут разрешаться в некотором смысле *не хуже*, чем во время сборки. Логично, что в обоих случаях используется загрузчик `ld.so`.

Рассмотрим теперь, как можно хранить типы символов. Ведь `debuginfo`-пакеты помещаются в отдельную компоненту репозитория, которая не доступна при сборке. Тогда можно было бы поместить отдельный файл с описанием символов в `devel`-пакет. Однако хотелось бы иметь полностью автоматическое решение. Поэтому предлагается хранить дополнительную информацию о символах в самих библиотеках, в специальной секции `.rpm.proto`. Эта секция даже не обязательно должна быть структурированной, как другие секции ELF; ее содержимое можно сжать, как это делается с секцией `minisymtab`. Для сжатия небольших текстовых файлов особенно хорошо подходит алгоритм `ppmd`.

В современной разновидности формата ELF, в которой используется секция `.gnu.hash`, экспортируемые символы в таблице `.dynsym` упорядочены по значению хеша, а неопределенные символы сгруппированы в начале таблицы [4, с. 9]. Это делает возможным особенно простое представление секции `.rpm.proto`: начиная с *первого экспортируемого символа*, записывать типы символов по одному на строку.

4 Дополнительная типизация

Дополнительная типизация не связана напрямую с символами. С ее помощью можно попытаться контролировать типы данных.

Пусть библиотека определяет какую-то структуру, а программа объявляет переменную этого типа. Тогда есть смысл контролировать размер этой структуры (так же, как при передаче структуры по значению). Структуру можно считать связанной с библиотекой, если 1) она используется в аргументах какой-либо функции библиотеки; 2) во время сборки библиотеки заголовочный файл с определением этой структуры устанавливается в `/usr/include`. Тогда библиотека предоставляет *псевдосимвол*, который описывает размер этой структуры.

Список литературы

[1] Алексей Турбин. Комплементарное хеширование подмножеств // Седьмая конференция разработчиков свободных программ. Тезисы докладов. М., 2010. С. 63–66.

[2] Felix Putze, Peter Sanders, Johannes Singler (2007)
Cache-, Hash- and Space-Efficient Bloom Filters

[3] Michael Matz et al. System V Application Binary Interface.
AMD64 Architecture Processor Supplement. Draft Version 0.99.7

[4] Ulrich Drepper. How To Write Shared Libraries
`www.akkadia.org/drepper/dsohowto.pdf`