Symbad: A New Tool Designed to Optimize Minimal Linux Distributions
Rodrigo J. Tobar\textsuperscript{1}, Mauricio A. Araya\textsuperscript{1}, Horst H. von Brand\textsuperscript{1}

Departamento de Informática, Universidad Técnica Federico Santa María, Avenida España 1680, Valparaíso, Chile

Abstract
symbad is a tool designed to scan and process the binary files of a Linux distribution in order to check their shared libraries dependencies and symbols. With this information, useful conclusions can be extracted, like discovering unused functions or even full unused libraries, among others. This information can be used for several purposes, such as optimizing the used disk space of a minimal Linux distribution or to get statistical usage information for a particular function. In this paper we present symbad and our experiences applying it over an One Laptop Per Child (OLPC) filesystem image.

1 Introduction

Linux is a widely used operating system kernel. Thanks to its GPL license [4], it has been used in a wide range of devices, including servers, personal computers and laptops. These last years it has become also very popular on even smaller devices, such as PDAs, cellphones and embedded systems [12]. The Linux versions running on those small devices are often based on major Linux distributions, like Fedora, Debian or Gentoo, just to mention a few. One example of this is the “One Laptop Per Child” (OLPC) project and its Sugar distribution, which is a fork of the early Fedora 7 Linux distribution. Like OLPC, there is a large number of “tiny” Linux distributions designed for small and embedded devices. One of the reasons why these distributions are “tiny” is because the devices where they are deployed have little storage, so the original distributions have to be shrunk to fit the requirements of the devices.

There are several points that must be considered when the purpose is shrinking a Linux distribution [9]. First, the kernel modules and driver support have to be patched and reduced to fit the device requirements. Over this layer, an application set is selected to run on the new distribution. With these steps the disk and RAM usage is lowered. Power and speed issues may have to be studied as well.

The current architecture for binary code execution on most operating systems is composed of executable binaries and shared libraries. Shared libraries are common code used by many independent executables. Executables make use of the code present in the shared libraries, which in turn are loaded into memory as required. Each executable file contains the list of the shared libraries it needs. This creates a complex dependency graph among the binaries on a system. This dependency graph is not precise at all, since not all the functions defined in a shared library are being actually used by the binaries to which it is linked. Because of this, the dependencies between binary files should not be seen as “a binary A depends totally on a binary library B”, but more like “binary A depends only on the functions implemented and exported in B that A uses” (see figure 1).

More detailed dependencies between binary files could be defined if we say “function f in binary A depends on function g defined in binary B” (see figure 2). This would give us better information about the usage of functions implemented on libraries. Alas, to be able to identify these dependencies the source code of the shared libraries and executable binaries would have to be inspected, which is a very complex task even on a open source distribution. Also, the benefits of doing this inspection may be minor compared to the effort spent on the task.

Here there is a new opportunity of optimization. If we consider all the dependencies between all the binaries that are present in a system, then we would be able to recognize which functions are implemented on a library, but never used by any other executable or library. Then, we could strip out these functions from the source files in order to produce “thinner” binaries and save disk (and memory) space, optimizing our system (see figure 3).
Figure 1: Dependencies between binary files should be seen not as a whole, but only on the functions that a binary imports from a second one.

Figure 2: A more detailed dependency graph can be drawn if we gather information of these dependencies at source code level.

Figure 3: A whole system presents multiple dependencies among its binary files. If all the dependencies are found, then the unused symbols of the libraries can be recognized and stripped. In the example above, the `setbuf` and `select` functions present in the `libc` library, and the `nan` and `sin` functions present in the `libm` library, are unused.

The same process could be used for various other purposes, like Linux distribution maintainers, package dependency construction, statistical purposes (which functions are the most used in order to improve them), and more. *symbad* aims to solve this problem by doing this checking automatically over a whole filesystem. Also, *symbad* could be used to analyze the complex dependencies of a specific multipackage software such an office suite, an ORB or a desktop environment.

### 2 Related Work

Currently there are several tools used to make binary analysis [6]. Depending on the way in which they analyze the files, these tools can be divided into two main categories: Static and dynamic. Dynamic binary analysis tools are software that analyze the code while it is being executed. Examples of these are debuggers (e.g., *gdb* [7]) and execution diagnostics (e.g., *strace* and *valgrind* [2]). These dynamic analysis tools are designed to solve runtime problems in the executables or libraries. On the other hand, static binary analysis tools analyze the code statically; that is, they don’t execute it but only read and analyze its internal structure. Among these tools we can find information displayers (e.g., *nm* and *objdump* [1]), symbol strippers (e.g., *strip* [1]) and more. Anyway, all these tools only
analyze one binary at a time, without giving much information about the relationships among them.

In terms of automatic dependency search, there is an old project called “Library Optimizer” [5]. It provides a tool called libopt, which resolves all shared library symbol references, rebuilding the shared libraries without the unused object files. The tool is coded in python, and makes use of the objdump and build utilities to get all the references between the system libraries, and finally rebuild the libraries without the unused functions. The project is abandoned since 2001, and it is missing functionality. It does not save the information that it processes, which makes it useless for statistical purposes or for further inspection.

3 Description of symbad

The problem described below cannot be solved manually, due to the sheer amount of data to process. Because of that, we have developed a software solution called symbad. symbad stands for “SYMbols of Binary Applications Database”, and it describes exactly what it is: A database where the information about the symbols (functions) belonging to the binary application files of a system are stored. The project started in November, 2006 by a group of students of the CSRG group at the UTFSM [3]. It is GPL licensed [4], and its development is maintained under the git versioning system [11]. symbad is a C application that scans a filesystem in search of all the existing binary application files. All these files are read to extract information about the symbols that are imported/exported by each of them. After this information is stored into a database, many queries can be done in order to obtain useful information from the stored data. At the time, Sqlite3 [10], MySQL [8] and PostgreSQL [14] database engines are supported by symbad, but extending the support for other database engines is simple to develop. For a long time now, Linux uses the Executable and Linking Format (ELF) as the specification of binary code files. Because of this, symbad uses the libelf library to get the information from the files through an ease to use API.

Figure 4: Data model of the symbad database. The grey-lined table is temporal, only used when filling and updating the database. The fields containing a sharp (#) conform the primary key.

The database used by symbad consists on six fixed tables, plus a temporal one (see figure 4). It works as follows: binnames contains the full path of a binary, along with its inode. The name is used as the primary key instead of the inode to prevent hard links to be ignored. This table serves as the index to all others. A similar table named tmp_binnames is created during the filling and updating process of the database, with the same information as above, used for comparisons and internal uses. binsyms stores all the imported symbols of the system, associating them to a binary file (executable or shared library). libsyms does the same, but with the exported symbols. For each of these two tables there is a corresponding index table (i.e., libindex and binindex), which store a summary of
the imported and exported symbols for each importer/exporter binary. Finally, libmap contains a map for the binary dependencies, indicating which libraries are needed by an importer binary.

### 3.1 symbad lifecycle

symbad has three main use cases. The first one has been described briefly below, and consists on the initial population of the database through the scanning of the filesystem in search of the binary application files located at it. The second action consists on updating the information placed in the database, which is done through a re-scan of the filesystem and a difference checking (delete old information, add new one). The third use case is when different kinds of queries are done to the database to obtain information from it.

#### 3.1.1 First database filling

When symbad is launched for the very first time, the only main action that can be taken is to fill the database. When filling the database, symbad scans a set of standard directories1 (or specified by an environment variable) in search of binary files. This scan results on a binary files list, where every entry is analyzed. Each binary file is read to get the information about the imported/exported symbols, along with the linked libraries. All this information is dumped into the database for further inspection.

#### 3.1.2 Updating the database

When updating the database, symbad also scans the whole filesystem. After that, the binary files present in the database and that are not in the generated file list are deleted from the database, and the newly found files are scanned and their information is stored in the database.

#### 3.1.3 Queries

The most called actions on symbad are, of course, the queries. Queries return useful information from the stored data (e.g., unused symbols, full unused libraries, linking dependencies). It might be interesting to write graphical clients that could retrieve this data from the database, in order to have a more graphical impression of the results.

Among all queries, the unused symbols count from a library is a special one. It requires an additional effort since the unused symbols count is not stored in the database. Because of this, whenever the query is asked for a given library, the database must be processed to get this information, saving it for future queries. If we want the information of the unused symbols for all the libraries, we should not execute this query library by library, but instead use an special option of symbad that post-processes all this information automatically, in order not to do so serially.

### 3.2 Generic database engine support

One of the main features of symbad is its support for multiple database engine. At the time of writing, Sqlite3 [10], MySQL [8] and PostgreSQL [14] engines are supported, and the addition of a new database engine can be done easily. For this, symbad uses a generic structure which contains pointers to the methods needed to open and close a database connection, along with a third one to execute commands on the database (see figure 5). This way, we have implemented three set of methods that adjust to this structure, and which can be chosen to work with one of the three database engine.

The five variables present in the structure correspond to internal data used by the methods to store: The database reference (db), private internal data (internal_data), error messages that an instruction could throw (error_msg), and the response values and count of the executed SQL statements (resv and resc, respectively). Depending on the configuration of symbad, the function pointers at the structure (db_open, db_close and db_exec) point to the corresponding functions that handle the connection to the database engine. This way, symbad can be extended in order to support more database engines in a simple way.

---

1/bin, /sbin, /lib64, /usr/bin, /usr/sbin, /usr/lib, /usr/lib64, /usr/games, /usr/kerberos, /usr/X11R6 and /usr/local
Figure 5: Generic structure used by symbad to support various database engines. Runtime configuration allows to chose one of the available database engines and use their functions to open and close the database, and execute SQL commands.

4 Results

symbad was originally developed to slim down the OLPC project. For this reason, symbad has been tested with the last available Sugar filesystem image, which is available at the OLPC ftp site [13]. The filesystem has a size of about 640 MiB (when unzipped). For the tests, the filesystem was unzipped and mounted via loopback on a local directory, on a PC with an Intel Core 2 Duo CPU @ 2.00GHz, 1 GiB on RAM. Over this OLPC filesystem, symbad scanned (recursively) the standard directories where binary files can be found. The database engine used was PostgreSQL 8.2 server running on the same machine.

With the described environment, symbad took about 299 seconds (almost 5 minutes) on the creation of the database. The post-processing task (optional) is more time consuming, longing about 1195 seconds (almost 20 minutes). The total time spent by symbad then, to perform both operation, was of 24 minutes and 54 seconds, which is a short time if we compare it with the time that the same task would have been done manually. Because the post-processing query has been done, all subsequent queries will take almost no time to be answered since new data processing does not need to be done. The database has a size of about 32 MiB after the whole process; this is, a 5% of the size of the scanned filesystem.

Regardless symbad is an automatized tool, there are special cases where the results may not reflect the reality, and that must be taken into account at the time of making conclusions.

- There are applications (e.g., the ssh and scp commands) that can be executed, but that are really ELF shared libraries. These files will get counted as libraries and, since no other binaries use the functions implemented in them, symbad will mark all these functions as unused.

- Plugins will be marked as unused as well. Since the applications into which they are plugged in are not directly linked with their plugins (they are loaded dynamically under the control of the running application), a clear dependency cannot be seen, so symbad will also make all the functions present in the plugins as unused.

Nevertheless, the most important information is that related to the unused symbols of the real libraries. We show in table 1 the unused symbols of the most referenced libraries of the system, as symbad tells us.

The total size of these 15 libraries is 18531178 bytes (about 18 MiB). If we consider all the functions of these libraries, we get that the average size of a function is 2738.46 bytes. If now we recocompile the libraries without the unused functions, and taking into account the average size of them, we would get a total size of 7530773 bytes, which is less than a half of the initial size of the libraries. Even when this is a rough calculation, it indicates how much space could be recovered in disk space (and in RAM, when the library is loaded into the main memory) if we remove unused symbols from the libraries. The same calculation can be made for the entire system. Alas, since symbad (currently) has the mentioned limitations (plugins and executable shared library recognition), it is impossible to get
Table 1: The 15 most used libraries on the OLPC filesystem image, their exported and unused symbols, and their sizes in bytes.

<table>
<thead>
<tr>
<th>Library</th>
<th>Exported</th>
<th>Unused</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/lib/librt.so.1</td>
<td>39</td>
<td>36</td>
<td>42040</td>
</tr>
<tr>
<td>/lib/i686/nosegneg/libc.so.6</td>
<td>1424</td>
<td>812</td>
<td>1682544</td>
</tr>
<tr>
<td>/lib/i686/nosegneg/libm.so.6</td>
<td>300</td>
<td>175</td>
<td>214580</td>
</tr>
<tr>
<td>/lib/i686/nosegneg/libpthread.so.0</td>
<td>163</td>
<td>87</td>
<td>127716</td>
</tr>
<tr>
<td>/lib/libdl.so.2</td>
<td>9</td>
<td>5</td>
<td>18804</td>
</tr>
<tr>
<td>/lib/libgobject-2.0.so.0</td>
<td>333</td>
<td>124</td>
<td>257544</td>
</tr>
<tr>
<td>/lib/libgmodule-2.0.so.0</td>
<td>8</td>
<td>1</td>
<td>9844</td>
</tr>
<tr>
<td>/lib/libpthread.so.0</td>
<td>2</td>
<td>1</td>
<td>15012</td>
</tr>
<tr>
<td>/lib/libglib-2.0.so.0</td>
<td>969</td>
<td>771</td>
<td>646316</td>
</tr>
<tr>
<td>/lib/libz.so.1</td>
<td>62</td>
<td>39</td>
<td>425773</td>
</tr>
<tr>
<td>/usr/lib/libselinux.so</td>
<td>166</td>
<td>120</td>
<td>91896</td>
</tr>
<tr>
<td>/lib/libgobject-2.0.so.0</td>
<td>333</td>
<td>124</td>
<td>257544</td>
</tr>
<tr>
<td>/usr/lib/libX11.so.6</td>
<td>767</td>
<td>562</td>
<td>4856705</td>
</tr>
<tr>
<td>/usr/lib/libxml2.so.2</td>
<td>1620</td>
<td>710</td>
<td>4856689</td>
</tr>
<tr>
<td>/usr/lib/libgstreamer-0.10.so.0</td>
<td>903</td>
<td>572</td>
<td>4859894</td>
</tr>
</tbody>
</table>

A second important result is the function reference statistics. Since symbad stores all the references between libraries and executables, the number of times that a function is required by any other library can be calculated (this is done with the post-processing option present in symbad). With this information we can build, for example, a list of the most used functions (see figure 6 and table 2). In the OLPC case, we have found that the 50 most referenced functions of the system all belong to the libc library, with free being the most used function of the system. As we mentioned in section 1, this does not correspond to the precise number of times that the function is being used in the
system, but to the number of libraries that make use of it. If the precise usage of the function is wanted, a dynamic code analysis would have to be done over the whole system.

<table>
<thead>
<tr>
<th>Function</th>
<th>Used times</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>pam_syslog</td>
<td>44</td>
<td>/lib/libpam.so.0.81.6</td>
</tr>
<tr>
<td>pam_get_user</td>
<td>29</td>
<td>/lib/libpam.so.0.81.6</td>
</tr>
<tr>
<td>is_selinux_enabled</td>
<td>28</td>
<td>/lib/libselinux.so.1</td>
</tr>
<tr>
<td>pam_get_item</td>
<td>26</td>
<td>/lib/libpam.so.0.81.6</td>
</tr>
<tr>
<td>Py_InitModule4</td>
<td>23</td>
<td>/usr/lib/libpython2.5.so.1.0</td>
</tr>
<tr>
<td>TIFFClose</td>
<td>23</td>
<td>/usr/lib/libtiff.so.3.8.2</td>
</tr>
<tr>
<td>pam_modutil_getpwnam</td>
<td>23</td>
<td>/lib/libpam.so.0.81.6</td>
</tr>
<tr>
<td>gst_debug_log</td>
<td>23</td>
<td>/usr/lib/libgstreamer-0.10.so.0.11.0</td>
</tr>
<tr>
<td>gst_element_get_type</td>
<td>23</td>
<td>/usr/lib/libgstreamer-0.10.so.0.11.0</td>
</tr>
<tr>
<td>freecon</td>
<td>21</td>
<td>/lib/libselinux.so.1</td>
</tr>
<tr>
<td>TIFFOpen</td>
<td>21</td>
<td>/usr/lib/libtiff.so.3.8.2</td>
</tr>
<tr>
<td>TIFFSetField</td>
<td>21</td>
<td>/usr/lib/libtiff.so.3.8.2</td>
</tr>
</tbody>
</table>

Table 2: The 12 most referenced functions of the system which are not in the libc library.

5 Conclusions

Linux distributions need to be minimized in order to be deployed in small devices. A critical point that is not often taken into account is the amount of symbols (functions) that the libraries of the system implement but that are totally unused. These functions use unnecessary disk space and RAM. symbad, through an automatic process, searches for these functions, and gives valuable information about the binary dependencies of our system.

Regardless the main purpose of symbad has been the detection of such symbols, it can be used for a large number of other tasks. Statistics of the most/least used functions on a system can be made; package maintainers can use the information stored by symbad to keep track of the dependencies between binary files, and so on.

Future work that must be done in symbad:

- Fully test symbad on 64 bit machines, with 64 bit ELF files. Filesystems containing both 64 and 32 bit binaries should be also supported. The database needs to be redesigned in order to support these two different ELF file types, and thus, having two versions (32 bits and 64 bits) of a symbol present in two different library files.

- New database engine support can be developed, in order to be able to connect to more databases and be more flexible for the users. Older versions of Sqlite could be supported.

- Locale information should be used to display the information in non-English languages, depending on the locale configuration under which symbad runs.

The actual state of symbad is more than sufficient for tests on a large set of machines, and to get very important information, useful for a wide range of objectives. Linux distribution maintainers are able to use this tool in its actual state, while people working with minimal distributions have a great resource that could help them optimize their work.

Acknowledgments

This work is part of the CSRG-UTFSM Group. This work could not have been possible without the help of users and contributors of symbad.
References


