Mining Debian Maintainer Scripts

Nicolas Jeannerod and Ralf Treinen
joint work with Yann Régis-Gianas

IRIF, Université Paris-Diderot

July 31, 2018
Plan

1. Intro
2. A First Step: A Static Parser for Shell Scripts
3. Statistical Analysis of Scripts
4. Findings
5. Conclusion
A `.deb` package contains two sets of files:

1. A set of files to install on the system when the package is installed,
2. And a set of files that provide additional metadata about the package or which are executed when the package is installed or removed. […] Among those files are the package maintainer scripts […]

(Debian Policy, introduction to ch. 3)
Different Maintainer Scripts

Roughly:

- **preinst** executed before the package is unpacked
- **postinst** executed after the package is unpacked
- **prerm** executed before the package is removed
- **postrm** executed after the package is removed
Breakdown by File Type

Sid amd64, as of 2018-05-23:

- 31.302 total (post|pre)(inst|rm)
- 10.737 are at least in part written by hand
- 31.048 POSIX shell
- 231 Bash
- 16 perl
- 5 ASCII (shell scripts without #! line)
- 2 ELF executables (preinst of bash and dash)
What Policy (Section 10.4) says

- Not required to be shell scripts
- csh and tcsh discouraged
- Should start on `#!`
- Should use `set -e`
- Posix standard 1-2017 with some embellishments:
  - `echo`, when built-in, must support `-n`
  - `test`, when built-in, must support `-a` and `-o`
  - local scopes
  - arguments to `kill` and `trap`
- We will focus on Posix(+debian)-shell scripts
Our goal

- Formal analysis of debian maintainer scripts
- Formal analysis *is not* testing: we aim at an assurance of correctness in any possible situation (program verification)
- Possible outcome: assertion of correctness (in an abstracted model), or detection of possible bugs.
- This talk: First findings from a syntactical analysis of maintainer scripts.
Why parsing POSIX shell is hard

- Designed for parsing and expanding on the fly
- Requires context-sensitive, and sometimes speculative parsing
- Words may be keywords according to context
- Assignment words are recognized depending on the context
- Here documents
- Actually undecidable in case of unrestricted use of `alias`
The Morbig parser for POSIX shell

- [https://github.com/colis-anr/morbig](https://github.com/colis-anr/morbig)
- Written in OCaml, uses the Menhir parser generator
- Speculative parsing and parse state introspection
- High-level code close to the POSIX specification
- See our presentation at FOSDEM’18 and minidebconf Hamburg’18
Concrete Syntax Trees produced by Morbig

type complete_command =
    | CompleteCommand_CList_Separator of clist' * separator'
    | CompleteCommand_CList of clist'
    | CompleteCommand_Empty

and complete_command_list = complete_command list

and clist =
    | CList_CList_SeparatorOp_AndOr of clist' * separator_op' * and_or'
    | CList_AndOr of and_or'

and and_or =
    | AndOr_Pipeline of pipeline'
    | AndOr_AndOr_AndIf_LineBreak_Pipeline of and_or' * linebreak' * pipeline'
    | AndOr_AndOr_OrIf_LineBreak_Pipeline of and_or' * linebreak' * pipeline'

...........

- types for concrete syntax trees (parse trees)
- corresponds directly to the grammar in the POSIX standard
- ~ 50 recursive type definitions
Visitors

- Imagine we want to code a tree traversal.
- 50 different types ⇒ we have to code 50 functions to traverse a syntax tree??
- The *visitor* design pattern comes to the rescue:
  - Visitors (iter, map, reduce, ...) are automatically generated thanks to a syntax extension (*libppx-visitors-ocaml-dev*)
  - Late Binding (as opposed to static binding) allows us to override only those of the functions that need to do interesting stuff.
A glimpse at the tool: **shstats**

- [https://github.com/colis-anr/shstats](https://github.com/colis-anr/shstats)
- works on the concrete syntax trees produced by morbig
- *expander* preprocessor attempts to expand parameters the values of which are statically known (see later).
- it is easy to add analyzer modules.
Example: find scripts with "" in words (1)

```ml
let options = [] and name = "dollar"
let dollar_scripts = ref ([]: string list)
let process_script filename cst =
  let detect_dollar =
    object (self)
      inherit [] Libmorbig.CST.reduce as super
      method zero = false
      method plus = (||)
      method! visit_word _env word =
        String.contains (UnQuote.on_string (unWord word)) '$'
    end
  in
  if detect_dollar#visit_complete_command_list () cst
  then dollar_scripts := filename::!dollar_scripts
```
Example: find scripts with "$" in words (2)

```ocaml
let output_report report =
  Report.add report
  "* Number of scripts with $ after expansion: %n\n"
  (List.length !scripts_with_dollar);
Report.add report "** Files:\n";
List.iter
  (function scriptname ->
    Report.add report
    "  - %s\n"
    (Report.link_to_source report scriptname))
  !scripts_with_dollar
```
Why tree traversal is useful here

- Counting occurrences of $ could have been done by grep ...
- Except for $ in comments, inside quotes, here documents without expansion, ...
- Tree traversal allows us to expand some of the variables
- More complicated things are possible, i.e. exclude variables of for loops.
Preprocessing: expand variable definitions when possible

```
1  x=1
2  if foo; then
3      y=2
4      echo $x $y
5  else
6      y=3
7      echo $x $y
8  fi
9  echo $x $y
```

Static expansion finds:
- line 4: x=1, y=2
- line 7: x=1, y=3
- line 9: x=1
So you think you understand assignments in shell?

Which value is printed by a script containing this fragment:

```bash
x=1
x=2 foo
echo $x
```

Possible choices:

1. 1
2. 2
3. 73
4. Syntax error
5. It depends
If that was too easy...

What does the following script print:

```sh
x=a
ox=b  y=${x:${z:=c}}  echo $x#$y#$z
echo $x#$y#$z
```

Missing `#!` line

- **Policy 10.4:**
  
  *All command scripts, including the package maintainer scripts inside the package and used by dpkg, should have a `#!` line naming the shell to be used to interpret them.*

- 39 offending packages in sid (November 2016)
- Bugs filed with severity *important*, after discussion at https://lists.debian.org/debian-devel/2016/11/msg00168.html
- 34 packages fixed by maintainer (July 2018)
Missing `set -e`

- Policy 10.4: 
  
  *Shell scripts (sh and bash) other than init.d scripts should almost certainly start with set -e...*

- 56 offending packages in sid (June 2017)

- Bugs filed with severity *normal*, after discussion at https://lists.debian.org/debian-devel/2017/06/msg00342.html

- 15 packages fixed by maintainer (July 2018)
Local

- Policy 10.4: \textit{local to create a scoped variable must be supported} [...] 

- However, \texttt{local} is not a nesting construction.

- This makes it in principle undecidable, for instance for an imaginary compiler, to know whether a variable is local.
Control structures

**local in a conditional**

```bash
f () {
    read line
    if [ $line = yes ]; then
        local x
    fi
    x=42
}
x=1
f
echo $x
```
Stats of `local` in maintainer scripts

Counting numbers of occurrences (not number of files):

- `local` outside of a function definition: 0
- `local` in a branching control structure (excluding function definitions inside a branch): 280
- `local` inside function definition, not in a branching structure: 2136
return outside function

install -o "USER" [...] || return 2

The Posix standard says:

The \textit{return} utility shall cause the shell to stop executing the current function or dot script. If the shell is not currently executing a function or dot script, the results are unspecified.

Should be:

install -o "USER" [...] || exit 2
# Most frequently used commands

<table>
<thead>
<tr>
<th>#</th>
<th>command</th>
<th>occ.</th>
<th>files</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[, test</td>
<td>57504</td>
<td>14832</td>
<td>47%</td>
</tr>
<tr>
<td>2</td>
<td>set</td>
<td>30687</td>
<td>30411</td>
<td>97%</td>
</tr>
<tr>
<td>3</td>
<td>true</td>
<td>15663</td>
<td>4532</td>
<td>14%</td>
</tr>
<tr>
<td>4</td>
<td>exit</td>
<td>14426</td>
<td>9183</td>
<td>29%</td>
</tr>
<tr>
<td>5</td>
<td>which</td>
<td>14423</td>
<td>13833</td>
<td>44%</td>
</tr>
<tr>
<td>6</td>
<td>echo</td>
<td>11427</td>
<td>5075</td>
<td>16%</td>
</tr>
<tr>
<td>7</td>
<td>dpkg-maintscript-helper</td>
<td>11113</td>
<td>3771</td>
<td>12%</td>
</tr>
<tr>
<td>8</td>
<td>rm</td>
<td>10779</td>
<td>7196</td>
<td>23%</td>
</tr>
<tr>
<td>9</td>
<td>dpkg</td>
<td>7633</td>
<td>7306</td>
<td>23%</td>
</tr>
<tr>
<td>10</td>
<td>deb-systemd-helper</td>
<td>6401</td>
<td>1409</td>
<td>5%</td>
</tr>
<tr>
<td>11</td>
<td>.</td>
<td>5194</td>
<td>3034</td>
<td>10%</td>
</tr>
<tr>
<td>12</td>
<td>grep</td>
<td>5039</td>
<td>4193</td>
<td>13%</td>
</tr>
<tr>
<td>13</td>
<td>db_get</td>
<td>4348</td>
<td>1252</td>
<td>4%</td>
</tr>
</tbody>
</table>
Most frequently used options

<table>
<thead>
<tr>
<th>opt.</th>
<th>occ.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e</td>
<td>30458</td>
<td>99.3%</td>
</tr>
<tr>
<td>-u</td>
<td>80</td>
<td>0.3%</td>
</tr>
<tr>
<td>-x</td>
<td>64</td>
<td>0.2%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opt.</th>
<th>occ.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>-f</td>
<td>8148</td>
<td>75.6%</td>
</tr>
<tr>
<td>-rf</td>
<td>1650</td>
<td>15.3%</td>
</tr>
<tr>
<td>-r</td>
<td>93</td>
<td>0.9%</td>
</tr>
</tbody>
</table>

Table: `set`

<table>
<thead>
<tr>
<th>opt.</th>
<th>occ.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>-L, --listfiles</td>
<td>6182</td>
<td>81.0%</td>
</tr>
<tr>
<td>--compare-versions</td>
<td>1261</td>
<td>16.5%</td>
</tr>
<tr>
<td>-s, --status</td>
<td>178</td>
<td>2.3%</td>
</tr>
</tbody>
</table>

Table: `rm`

Table: `dpkg`
Invalid command option

```
mkdir -f /etc/foobar &> /dev/null || true
```

Should be:

```
mkdir -p /etc/foobar
```
## Frequency of unary test operators

<table>
<thead>
<tr>
<th>operator</th>
<th>occurrences</th>
<th></th>
<th>operator</th>
<th>occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x</td>
<td>9480</td>
<td></td>
<td>-r</td>
<td>600</td>
</tr>
<tr>
<td>-d</td>
<td>5488</td>
<td></td>
<td>-h</td>
<td>295</td>
</tr>
<tr>
<td>-e</td>
<td>5317</td>
<td></td>
<td>-c</td>
<td>20</td>
</tr>
<tr>
<td>-n</td>
<td>3767</td>
<td></td>
<td>-S</td>
<td>8</td>
</tr>
<tr>
<td>-f</td>
<td>3239</td>
<td></td>
<td>-w</td>
<td>5</td>
</tr>
<tr>
<td>-z</td>
<td>1900</td>
<td></td>
<td>-p</td>
<td>4</td>
</tr>
<tr>
<td>-s</td>
<td>838</td>
<td></td>
<td>-b</td>
<td>2</td>
</tr>
<tr>
<td>-L</td>
<td>755</td>
<td></td>
<td>-u</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-k</td>
<td>1</td>
</tr>
</tbody>
</table>
## Frequency of binary test operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>27981</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>1393</td>
</tr>
<tr>
<td><code>-eq</code></td>
<td>185</td>
</tr>
<tr>
<td><code>-gt</code></td>
<td>179</td>
</tr>
<tr>
<td><code>-ne</code></td>
<td>65</td>
</tr>
<tr>
<td><code>-le</code></td>
<td>51</td>
</tr>
<tr>
<td><code>-lt</code></td>
<td>32</td>
</tr>
<tr>
<td><code>-ge</code></td>
<td>19</td>
</tr>
<tr>
<td><code>-ef</code></td>
<td>7</td>
</tr>
<tr>
<td><code>-nt</code></td>
<td>2</td>
</tr>
</tbody>
</table>
Usage of \(-a\) and \(-o\) in tests

- In sid: 2467 occurrences in 1850 scripts
- Mandated by Policy 10.4: 
  \[
  \text{test, if implemented as a shell built-in, must support } \neg a \text{ and } \neg o \text{ as binary logical operators.}
  \]
- POSIX: \(-a\) and \(-o\) are an obsolete extension.
- The GNU info page says:
  \[
  \text{Note it’s preferred to use shell logical primitives rather than these logical connectives internal to ‘test’, because an expression may become ambiguous depending on the expansion of its parameters.}
  \]
Ambiguity of test expressions

■ Stems from the fact that single word \( w \) is a valid test (checking whether the word is non-empty).

■ Example: \( ( = ) \) (maybe obtained from \( ( \$1 = \$2 ) \))

■ Example: What should be the result of

\[
[ -a -a -a -a -a ]
\]
\[
echo $?
\]

■ Different results by different shells:

- dash 0
- bash 1
- bash -posix 1
How to avoid \texttt{-a and \texttt{-o}}

Both POSIX and GNU recommend to replace

\begin{verbatim}
  test EXPR1 -a EXPR2
  test EXPR3 -o EXPR4
\end{verbatim}

by

\begin{verbatim}
  test EXPR1 && test EXPR2
  test EXPR3 || test EXPR4
\end{verbatim}
Syntax errors in test expressions

- An error of test in the condition of an *if-then-else* or a *while* loop is seen by the shell as the value *false* (strict mode is temporarily disabled)
- Found 9 errors (June 2018)
- Bugs filed with varying severity
Examples of mistakes in test expressions (1)

`if [ pathfind "foobar" = 0 ]; then`

Should be:

`if [ $(pathfind "foobar") = 0 ]; then`
Examples of mistakes in test expressions (2)

```bash
if [ "$1" = "remove" ] || \
[ "$1" = "disappear" ] || [ "$1" = "purge" ] ; then

Should be:

```bash
if [ "$1" = "remove" ] || \
[ "$1" = "disappear" ] || [ "$1" = "purge" ] ; then
```
Examples of mistakes in test expressions (3)

```bash
if [ "$1" != "upgrade" ]; then

Should be:

if [ "$1" != "upgrade" ]; then
```
Examples of mistakes in test expressions (4)

```bash
if [ /etc/jabber-querybot/Querymodule.pm -ef /
/usr/share/doc/jabber-querybot/examples/Testbot.pm ];

Should be:

```bash
if [ /etc/jabber-querybot/Querymodule.pm -ef \
/usr/share/doc/jabber-querybot/examples/Testbot.pm ];
```
Examples of mistakes in test expressions (5)

if [ "$2" \< "1.2-3.4" ];

Should (probably) be

if dpkg --compare-versions "$2" lt "1.2-3.4";
Questionable Redirections

```bash
foo --verbose --help 2>&1 >/dev/null
```

Should be:

```bash
foo --verbose --help >/dev/null 2>&1
```

- 124 occurrences of that problem
- MBF: to be discussed
Also: Useless Redirections

```bash
echo "foo $name bar" >&1

echo postinst "$1" >&2 >/dev/null
```
The CoLiS Project

- Correctness of Linux Scripts
- Project funded by Agence Nationale de Recherche

- October 2015 – September 2020
- http://colis.irif.fr/
- Future work: tree transducer (team at INRIA Lille), symbolic execution (teams at INRIA Saclay and Univ. Paris-Diderot).