System-Wide Elimination of Dynamic Symbols

Vladislav Ivanishin  Evgeny Kudryashov  Alexander Monakov
Dmitry Melnik  Jehyung Lee

ISP RAS

November 30, 2017
Background

Lots of “dead weight” in shared libraries:

- Obsolete interfaces (gets())
- Very rarely used APIs (<complex.h>)
- Backwards compatibility
Background

Lots of “dead weight” in shared libraries:

▶ Obsolete interfaces (`gets()`)
▶ Very rarely used APIs (`<complex.h>`)  
▶ Backwards compatibility

No need for most of that on special-purpose hardware

▶ Special-purpose distros (Android, Tizen)
▶ Small-devices (IoT) Tizen profile
Problem Statement

Slim down by eliminating unused code/data from shared libraries
Possible? Not in general:
  ▶ Future applications may use any public API
  ▶ Binaries may use any backward-compat API
Assume “closed world” full-distro rebuilds
  ▶ Can see what APIs get used
  ▶ No “potential future uses”
Aside: Elimination in Static Linking

For static linking, already available in practice:

1. Compile with `gcc -ffunction-sections -fdata-sections`:
   
   Per-function sections
   
   ```
   .section .text.foo,"ax",@progbits
   .globl foo
   .type foo, @function
   foo:
   movl $42, %eax
   ret
   ```

2. Link with `-gc-sections`
   
   Linker omits sections not reachable by relocations from the entry point
-gc-sections for Dynamic Modules

Can we use -gc-sections for shared libraries?
For dynamic linking, entrypoint is not the only GC root

▶ The .dynamic section is another root
  Points to dynamic symbols and global library constructors/destructors

▶ Most code is reachable from dynamic symbols (the library’s interface)

▶ Reducing the API surface (changing symbol’s visibility to “hidden”) allows GC
Dynamic Dependencies

Want to compute reachability on dynamic symbol set

- Explicit dependencies

**Direct Call**

```c
int main()
{
    puts("Hello World");
}
```
Dynamic Dependencies

Want to compute reachability on dynamic symbol set

- Explicit dependencies
- Implicit, via dlsym()

Dynamic Lookup via dlsym

```c
#include <dlfcn.h>

void *dlsym(void *handle, const char *name);

void malloc(size_t n)
{
    void *real_malloc = dlsym(RTLD_NEXT, "malloc");
    ...
}
```
Dynamic Dependencies

Want to compute reachability on dynamic symbol set
  ▶ Explicit dependencies
  ▶ Implicit, via dlsym()
  ▶ Non-standard runtime lookups

Dynamic Lookup via direct inspection

```
#include <elf.h>
extern Elf64_Dyn _DYNAMIC[];

...
Elf64_Dyn *dyn = ...;
for (int i = 0; dyn[i] != DT_NULL; i++)
  if (dyn[i].d_tag == DT_SYMTAB) {
    ...
  }
...
```
Dynamic Dependencies

Want to compute reachability on dynamic symbol set

▶ Explicit dependencies
▶ Implicit, via dlsym()
▶ Non-standard runtime lookups

Must be conservative

▶ Missed explicit deps — build-time link error
▶ Missed runtime deps — run-time fallback or error
Handling Runtime Dependencies

Targets of runtime lookups impossible to compute exactly:

- Custom, non-dlsym lookups impossible to analyze
- Issues with dlsym lookups:
  - dlsym wrappers
  - Non-constant name argument
  - Interpreters may expose dlsym to scripts (Lua, Python)

However, calls to dlsym are statically visible

- Discover wrappers for dlsym
- Best-effort analysis of dlsym-like functions
- Allow manual annotation where static analysis fails
- Completely punt on low-level lookups
High-level Approach

1. Discover dlsym wrappers
2. Try to compute dlsym lookup targets
3. Record static dependencies
4. Analyze distro-wide symbol dependency graph
5. Eliminate unused symbols

Three distro-wide rebuilds required in total

Implementation goals:

- GCC plugins for dlsym analysis
- Linker plugins for explicit deps and elimination
Annotating dlsym Wrappers

Constraints/assumptions:

▶ Wrappers may be used across translation units
▶ Link-time analysis not sufficient
▶ Symbol \textit{name} argument passed unchanged

Solution: two-stage algorithm:

1. Dump all \textit{jump functions} distro-wide

Definition (Jump functions)

GCC IPA term for function call argument transfer

If we have

```c
void *next(const char *name) { return dlsym(RTLD_NEXT, name); }
```

then we have a graph edge dlsym/1 \rightarrow next/0
Annotating dlsym Wrappers

Constraints/assumptions:

- Wrappers may be used across translation units
- Link-time analysis not sufficient
- Symbol *name* argument passed unchanged

Solution: two-stage algorithm:

1. Dump all *jump functions* distro-wide
2. Compute *transitive closure* on jump function graph from root *dlsym/1*
Computing dlsym Targets

Multiple targets per one dlsym call site

```c
const char *ICU_API[] = {"ucol_open", "ucol_close", ...};
...
  for (i = 0; i < ICU_FUNC_CNT; i++) {
    handle = dlsym(g_dl_icu_handle, ICU_API[i]);
    ...  
    icu_handle[i] = handle;
  }
```

In the compiler plugin:

- Iteration over callsites of dlsym-like functions
- Simple GIMPLE analysis for array/struct references
Recording Static Dependencies

Use LTO plugin interface for introspection
  ▶ Avoid patching the linker
  ▶ Avoid duplicated work (search in static archives)

The `claim_file_handler` API hook allows to inspect object files
  ▶ Find symbol tables
  ▶ Find relocation tables
  ▶ Resolved relocations give intra-DSO dependencies
  ▶ Cross-DSO deps are given by dynamic relocations
Eliminating Unused Symbols

Two opportunities: compile time (in GCC) and link time

1. At compile time: optional, for optimization
   ▶ Compiler doesn’t process asm inputs
   ▶ More constrained

2. At link time: required
   ▶ Can eliminate more than the compiler

Implementation:

1. Force-enable \texttt{-gc-sections}

2. Set \textit{hidden visibility} on eliminated symbols
   Linker plugin makes copies of .o files with adjusted visibility info
Identifying Matching Object Files

Need to robustly identify translation units

- Names can be too common: conftest.c
- ...or unstable: /tmp/cc123abc.o

Generate a stable, unique srcid for each object file

- Strong hash of blinded IR dump before optimizations
- “Blinding” removes string contents: stabilize against __DATE__ substitutions
- Emit an empty .comment.privplugid.srcid section
Pitfalls

*Global* transformation proved to be problematic

- Must ensure all dependencies seen in analysis
- Must be able to rebuild all packages with elimination

Issues with driving elimination from linker plugin

- Some projects only build with `ld.bfd` (Glibc)
- Linker plugin API underspecified
- Not interoperable with LTO
- Versioned symbols poorly supported
- BFD linker incorrectly orders shared libraries with plugin
## Results: Aggregate Section Sizes before/after Optimization

<table>
<thead>
<tr>
<th>section name</th>
<th>original size, KB</th>
<th>optimized size, KB</th>
<th>delta, KB</th>
<th>delta, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>30211</td>
<td>24176</td>
<td>6035</td>
<td>19</td>
</tr>
<tr>
<td>.data</td>
<td>310</td>
<td>293</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>.data.rel.ro</td>
<td>262</td>
<td>233</td>
<td>29</td>
<td>11</td>
</tr>
<tr>
<td>.rodata</td>
<td>5831</td>
<td>4701</td>
<td>1130</td>
<td>19</td>
</tr>
<tr>
<td>.dynstr</td>
<td>1885</td>
<td>859</td>
<td>1025</td>
<td>54</td>
</tr>
<tr>
<td>.dynsym</td>
<td>1347</td>
<td>860</td>
<td>486</td>
<td>36</td>
</tr>
<tr>
<td>.got</td>
<td>248</td>
<td>192</td>
<td>56</td>
<td>22</td>
</tr>
<tr>
<td>.plt</td>
<td>610</td>
<td>452</td>
<td>158</td>
<td>25</td>
</tr>
<tr>
<td>.rel.dyn</td>
<td>415</td>
<td>356</td>
<td>59</td>
<td>14</td>
</tr>
<tr>
<td>.rel.plt</td>
<td>397</td>
<td>292</td>
<td>105</td>
<td>26</td>
</tr>
<tr>
<td>.hash</td>
<td>594</td>
<td>383</td>
<td>211</td>
<td>35</td>
</tr>
</tbody>
</table>

Thank you!