Pruning ELF: Size Optimization of Dynamic Shared Objects at Post-link Time

Vladislav Ivanishin Evgeny Kudryashov Alexander Monakov
Dmitry Melnik Jehyung Lee

ISPRAS

November 22, 2018

Problem Statement

Given: a distribution with **shared libraries**, **immutable** once it's built (i.e. no package manager).

Slim it down by eliminating unused code/data

Problem Statement

Given: a distribution with **shared libraries**, **immutable** once it's built (i.e. no package manager).

Slim it down by eliminating unused code/data

assuming "closed world" full-distro rebuilds

- No packages bypass the toolchain we control
- Nothing is added afterwards; 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

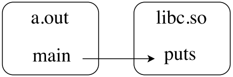
Dependency Types

Want to compute reachability on dynamic symbol set

- Link-time dependencies

Direct Call

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



Dependency Types

Want to compute reachability on dynamic symbol set

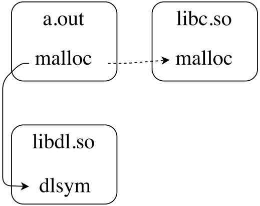
- Link-time dependencies
- Run-time dependencies via dlsym()

Dynamic dlsym Lookup

```
#include <dlfcn.h>

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

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



Dependency Types

Want to compute reachability on dynamic symbol set

- Link-time dependencies ← this talk only covers this kind
- Run-time dependencies via `dlsym()` ← described in [1]
- Other run-time dependencies ← only manual annotation

High-level Approach

1. Record link-time dependencies (requires whole system rebuild)
2. Analyze system-wide symbol dependency graph
3. Eliminate unused symbols (another whole system rebuild)

Recording Link-time Dependencies

Use LTO plugin interface for introspection

The `claim_file_handler` API hook allows to inspect object files and extract necessary info

Analyzing System-wide Dependency Graph

- stand-alone tool
- takes dependencies collected at the previous step from all links
- merges them into one global graph
  $V = \{\text{sections and symbols}\}$, $E = \{\text{relocations and definitions}\}$
- traverses it from entry points

Eliminating Unused Symbols, Prior Approach

Idea: eliminate at link time. Compared to compile-time:

- Required: arbitrary source language
- Elimination on per-DSO basis

Implementation:

1. Force-enable `--gc-sections`
2. Set *hidden visibility* on eliminated symbols. Tried 2 methods:
    - Linker plugin claims the input `.o` files and adds their copies with adjusted visibility info to the link (via `add_input_file`)
    - Auxiliary `.o` file with references to convey visibility info

Eliminating Unused Symbols, Prior Approach: Problems

- Probing done by `configure` scripts—have to be conservative
- `configure` divergence is hard to track and not user-friendly
- Various linker bugs (plugin API and `--gc-sections` in combo with visibility rules are not among the best tested features)

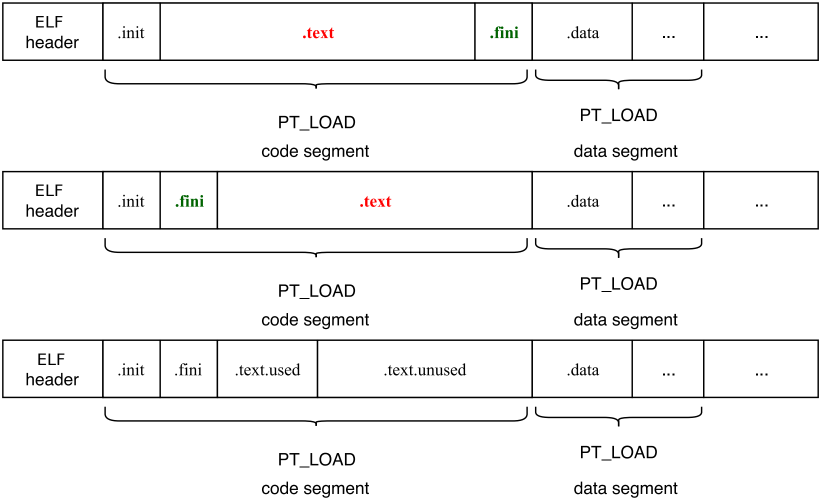
Eliminating Unused Symbols, New Approach

Idea: binary post-processing

- Divide loadable segments into used/unused, chop off the tails (This requires link-time section reordering—e.g. with a plugin)
- Regenerate associated tables

Cross-segment references are fine, because virtual addresses are not modified.

Eliminating Unused Symbols, New Approach



ELF header | .init | .text | .fini | .data | ... | ...

PT_LOAD code segment

PT_LOAD data segment

ELF header | .init | .fini | .text | .data | ... | ...

PT_LOAD code segment

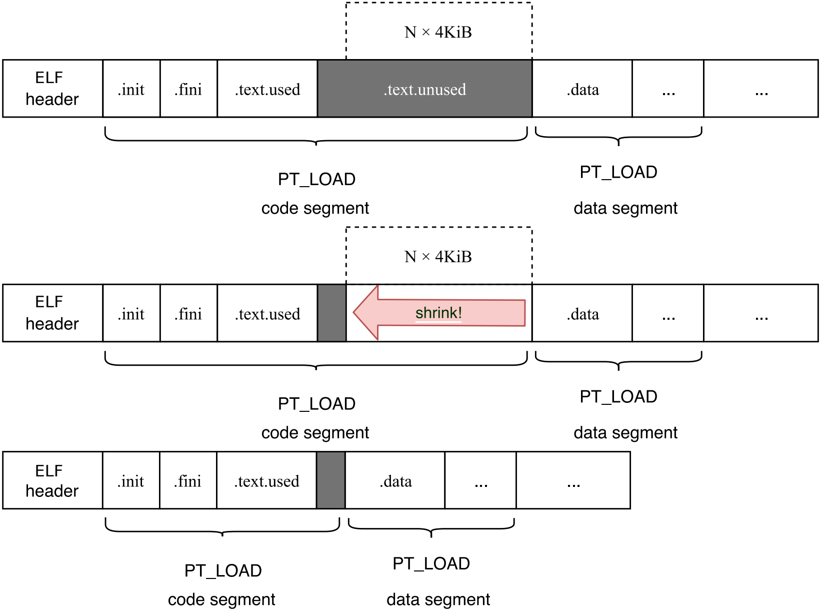
PT_LOAD data segment

ELF header | .init | .fini | .text.used | .text.unused | .data | ... | ...

PT_LOAD code segment

PT_LOAD data segment

Eliminating Unused Symbols, New Approach



Eliminating Unused Symbols, New Approach

4K problem mitigation:

```
.text.used [.text.unused .data.unused] .data.used
```

Tables:

- `.hash`, `.dynsym`: regenerate
- `.dynstr`: regenerate (suffix merging)
- `.got`, `.plt`: leaving works but wastes space, regenerating is problematic due to resolved references and the 4K problem

Most of the tables can be emitted to a separate segment.

Eliminating Unused Symbols, New Approach

Pros:

- better reproducibility: configure tests at step 3 will probe unmodified (modulo reordering) binaries, same as at step 1
- potential to eliminate more: no need to consider `mains` of configure tests as roots for reachability analysis
- doesn't suffer from any linker bugs (related to `--gc-sections`, versioned symbols, or plugin API implementation)

Cons/limitations:

- requires `LDPT_UPDATE_SECTION_ORDER` plugin interface which is only implemented in Gold
- and a small patch for Gold (move `ORDER_FINI`, `ORDER_EHFRAME` above `ORDER_TEXT`)
- hard to regenerate and shrink `.dynstr`, `.plt` (and references to them), and hash tables (not done in our PoC implementation)
- 4K alignment overhead (missed optimization) per DSO

The Code

This project is free software and is available from

`https://github.com/ispras/libosuction`

(branch `vlad/segshrink-v6`)

Bibliography

- V. Ivanishin, E. Kudryashov, A. Monakov, D. Melnik, and J. Lee.
  System-wide elimination of unreferenced code and data in dynamically linked programs.
  In *2017 Ivannikov ISPRAS Open Conference (ISPRAS)*.

- V. Ivanishin, E. Kudryashov, A. Monakov, D. Melnik, and J. Lee.
  Pruning ELF: Size optimization of dynamic shared objects at post-link time.

- (Ab)using LTO plugin API for system-wide shrinking of dynamic libraries.
  GNU Tools Cauldron 2018.

- Ian Lance Taylor.
  Linkers.
  `https://lwn.net/Articles/276782/`.

- WHOPR driver design.
  `https://gcc.gnu.org/wiki/whopr/driver`.