ELF linking: what it means and why it matters

Stephen Kell

stephen.kell@cl.cam.ac.uk

joint work with Dominic P. Mulligan and Peter Sewell

Computer Laboratory
University of Cambridge

A kernel is born

```
ld -m elf_x86_64 --build-id -o vmlinux \
 -T arch/x86/kernel/vmlinux.lds \
 arch/x86/kernel/head{_64,64,}.o \
 arch/x86/kernel/init_task.o init/built-in.o \
 --start-group \
 {usr,arch/x86,kernel,mm,fs}/built-in.o \
 {ipc,security,crypto,block}/built-in.o \
 lib/lib.a arch/x86/lib/lib.a \
 lib/built-in.o arch/x86/lib/built-in.o \
 {drivers,sound,firmware}/built-in.o \
 {arch/x86/{pci,power,video},net}/built-in.o \
 --end-group \
 .tmp_kallsyms2.o
```

How can we get strong guarantees about software like this?

Shopping list

- specify the architecture(s)
- specify the C source language
- verify the compiler
- specify & verify the hardware
- specify & verify functional properties…

All good stuff, but

- what was actually happening in that link command?
- … something we can hand-wave away, right?

Of POPLs past (1)

1 Introduction

Program modularization arose from the necessity of splitting large programs into fragments in order to compile them. As system libraries grew in size, it became essential to compile the libraries separately from the user programs; libraries acquired interfaces that minimized compilation dependencies. A linker was used to patch compiled fragments together.

Cardelli
"Program Fragments, Linking and Modularization"
POPL '97

Of POPLs past (2)

and compilable. In this paper we provide a context where linking can be studied, and separate compilability can be formally stated and checked. We propose a framework where each module is separately compiled to a self-contained entity called a *linkset*; we show that separately compiled, compatible modules can be safely linked together.

Is separate compilation really the substance of linking?

- hint: no

That kernel again

```
ld -m elf_x86_64 --build-id -o vmlinux \
 -T arch/x86/kernel/vmlinux.lds \
 arch/x86/kernel/head{_64,64,}.o \
 arch/x86/kernel/init_task.o init/built-in.o \
 --start-group \
 {usr,arch/x86,kernel,mm,fs}/built-in.o \
 {ipc,security,crypto,block}/built-in.o \
 lib/lib.a arch/x86/lib/lib.a \
 lib/built-in.o arch/x86/lib/built-in.o \
 {drivers,sound,firmware}/built-in.o \
 {arch/x86/{pci,power,video},net}/built-in.o \
 --end-group \
 .tmp_kallsyms2.o
```

Another shopping list

1. specify the object file formats
2. specify the linker's own language(s!)
3. verify the linker
4. go back to the other shopping list

The rest of this talk: our start on tackling these.

- non-idealised spec of Unix linking
- … ELF object format…
- … and (static) linking of ELF files
- ambition: usable as test oracle

+ some experience from a "systems person"

Systems software is written in. . .

. . . in C, mostly, right? With a bit of assembly?

Systems software is written in. . .

. . . in C, mostly, right? With a bit of assembly?

```
/* NOTE: gcc doesn't actually guarantee that global objects will be
 * laid out in memory in the order of declaration, so put these in
 * different sections and use the linker script to order them. */
```

Systems software is written in. . .

. . . in C, mostly, right? With a bit of assembly?

```
/* NOTE: gcc doesn't actually guarantee that global objects will be
 * laid out in memory in the order of declaration, so put these in
 * different sections and use the linker script to order them. */
pmd_t pmd0[PTRS_PER_PMD] __attribute__ ((
  __section__ (".data..vm0.pmd"), aligned(PAGE_SIZE)));
pgd_t swapper_pg_dir[PTRS_PER_PGD] __attribute__ ((
   __section__ (".data..vm0.pgd"), aligned(PAGE_SIZE)));
pte_t pg0[PT_INITIAL * PTRS_PER_PTE] __attribute__ ((
   __section__ (".data..vm0.pte"), aligned(PAGE_SIZE)));
```

Systems software is written in. . .

. . . in C, mostly, right? With a bit of assembly?

```
/* NOTE: gcc doesn't actually guarantee that global objects will be
 * laid out in memory in the order of declaration, so put these in
 * different sections and use the linker script to order them. */
pmd_t pmd0[PTRS_PER_PMD] __attribute__ ((
  __section__ (".data..vm0.pmd"), aligned(PAGE_SIZE)));
pgd_t swapper_pg_dir[PTRS_PER_PGD] __attribute__ ((
   __section__ (".data..vm0.pgd"), aligned(PAGE_SIZE)));
pte_t pg0[PT_INITIAL * PTRS_PER_PTE] __attribute__ ((
   __section__ (".data..vm0.pte"), aligned(PAGE_SIZE)));
```

Semantically, this is crucial!

It's this whole other language

```
/* Put page table entries (swapper_pg_dir) as the first thing
 * in .bss. This ensures that it has bss alignment (PAGE_SIZE). */
. = ALIGN(bss_align);
.bss : AT(ADDR(.bss) - LOAD_OFFSET) {
  *(.data..vm0.pmd) *(.data..vm0.pgd) *(.data..vm0.pte)
  *(.bss..page_aligned)
  *(.dynbss) *(.bss)
  *(COMMON)
}
```

Command lines are languages too

```
Usage: /usr/local/bin/ld.bfd [options] file...
Options:
  -e ADDRESS, --entry ADDRESS  Set start address
  -E, --export-dynamic         Export all dynamic symbols
  -O                           Optimise output file
  -r, -i, --relocatable        Generate relocatable output
  -R FILE, --just-symbols FILE Just link symbols
  -T FILE, --script FILE       Read linker script
  -(, --start-group            Start a group
  -), --end-group              End a group
  --as-needed                  Only set DT_NEEDED for following d
  -Bstatic, -dn, -static       Do not link against shared librari
  -Bsymbolic                   Bind global references locally
  --defsym SYMBOL=EXPRESSION   Define a symbol
  --gc-sections                Remove unused sections (on some ta
  --sort-section name|align    Sort sections by name or maximum a
```

Doesn't this matter only for obscure systems code?

```
void *malloc(size_t sz)
{ /* my own malloc */ }

int main(void)
{ // ...
  int *is = malloc(42 * sizeof (int));
}
```

Will it call my malloc() or the "other" one? Depends:

- statically or dynamically linked?
- what linker options?
- what compiler options?
- where does the other malloc() come from?

Linker-speak: what it's used for

- memory layout
- memory placement
- inter-module encapsulation
- inter-module binding
- inter-module versioning
- link-time deduplication
- build-time flexibility & configuration
- extensibility
- instrumentation
- introspection
- …

Linker-speak: where it's specified

- early Unix documentation
- man pages
- folklore
- source code
- the minds of hackers

One good linker deserves another

- 1972: AT&T Unix linker
- 1977: BSD linker
- c.1983: original GNU linker
- 1988: System V r4 linker (introduces ELF)
- c.1990: GNU BFD linker
- 2008: GNU `gold` linker
- c.2012: LLVM `lld` linker

A common ambition

- be “mostly like that other linker”
- can I link my programs yet? do they seem to work?

Other platforms are available…

Back to the kernel

```
ld -m elf_x86_64 --build-id -o vmlinux \
 -T arch/x86/kernel/vmlinux.lds \
 arch/x86/kernel/head{_64,64,}.o \
 arch/x86/kernel/init_task.o init/built-in.o \
 --start-group \
 ... # snip
```

Questions we could ask:

- does the output binary do the right thing?
- are we using the linker the right way [for that]?
- did the linker do its job correctly?

Back to the kernel

```
ld -m elf_x86_64 --build-id -o vmlinux \
 -T arch/x86/kernel/vmlinux.lds \
 arch/x86/kernel/head{_64,64,}.o \
 arch/x86/kernel/init_task.o init/built-in.o \
 --start-group \
 ... # snip
```

Questions we could ask:

- does the output binary do the right thing?
- are we using the linker the right way [for that]?
- **did the linker do its job correctly?**

First step: executable spec for an ELF *static* linker

Lem spec of ELF static linking

- ELF file format
- executable, actually working linker!
- architectures: x86-64 and partial AArch64, PPC64
- readable! comments, factoring

About 2 person-years of effort so far…

What it can do

Link small programs against a small/real libc (uClibc)

- hello, bzip2, …
- GNU C library exercises a *lot* of linker features
  - “almost works”

Next step: link checker

- take a link job + output, answers y/n
- challenge: accommodate looseness
- ordering, padding, merging, discarding, relax / opt …

What's involved

- read command line
- gather input files (incl. archives, scripts)
- resolve symbols
- discard unneeded inputs
- size support structures (GOT, PLT, . . .)
- interpret linker script. . .
- . . . one pass to define & size output
- . . . another pass to place output
- complete support structures
- apply relocations
- write output file

A specification of sorts

```
ld -o OUTPUT /lib/crt0.o hello.o -lc
```

- `-lc` maps to the archive `libc.a`

> The linker will search an archive only once, at the location where it is specified on the command line. If the archive defines a symbol which was undefined in some object which appeared before the archive on the command line, the linker will include the appropriate file(s) from the archive. However, an undefined symbol in an object appearing later on the command line will not cause the linker to search the archive again.

Other linkers sometimes do something slightly different...

A more precise specification

```
let  def_is_eligible  = (fun (*  ...  *) ->
  let  (*  snip  more supporting definitions  ...  *)
  in
  let  ref_and_def_are_in_same_archive
   = match (def_coords, ref_coords)  with
    (InArchive(x1,  _)  ::  _,  InArchive(x2,  _)  ::  _)  -> x1 = x2
    |  _  -> false
  end in
  (*  main  eligibility  predicate  *)
  if  ref_is_defined_or_common_symbol then def_sym_is_ref_sym
  else
     if  ref_is_unnamed then false (*  never match empty names  *)
    else
       if  def_in_archive  <> Nothing then
```

Is that enough? Is it correct?

ELF file format spec is quite well validated.

Linking spec is not quite a complete spec of real linking

- some looseness (e.g. in link order) not captured yet
- ABI-specific optimisations not modelled

$\rightarrow$ not *yet* usable as test oracle, but not far off…

More than a reference implementation

- … capture space of permitted links
- usable in proof

Use in proof

- extracted to Isabelle/HOL (33,150 lines)
- proved termination of linker on all inputs
  - (around 1,500 lines)
- proved a sample correctness theorem
  - about (very simple) relocation on AMD64
  - around 4,500 lines
  - … mostly re-usable lemmas

Reflections of a systems hacker

Getting used to functional style is no biggie. But

- can't forget performance
- tool maturity matters
- linguistic convenience matters
- type-theoretic errors/problems can be inscrutable
  - even to the fp-competent

Example: labelled memory images (1)

Our "intermediate representation"!

```
(* An element might have an address/offset, and
 * it has some contents. *)
type element = <| startpos : maybe natural
                ; length   : maybe natural
                ; contents : byte_pattern
                |>
type memory_image = Map.map string element (* name -> content *)
```

Example: labelled memory images (2)

```
type range = natural * natural  (* start, length *)
type element_range = string * range (* element id, range *)

type annotated_memory_image 'abifeature = <|
    elements : memory_image
  ; by_range : set ((maybe element_range) * (range_tag 'abifeature))
  ; by_tag   : multimap (range_tag 'abifeature) (maybe element_range)
  |>
```

Roll your own

- identity (gensym)
- ordering

The horror

```
let elfFileFeatureCompare f1 f2 =
    match (f1, f2) with
        (ElfHeader(x1), ElfHeader(x2)) -> (* equal tags, so ... *) compare x1 x
      | (ElfHeader(x1), _) -> LT
      | (ElfSectionHeaderTable(x1), ElfHeader(x2)) -> GT
      | (ElfSectionHeaderTable(x1), ElfSectionHeaderTable(x2)) -> (* equal tag
      | (ElfSectionHeaderTable(x1), _) -> LT
      | (ElfProgramHeaderTable(x1), ElfHeader(x2)) -> GT
      | (ElfProgramHeaderTable(x1), ElfSectionHeaderTable(x2)) -> GT
      | (ElfProgramHeaderTable(x1), ElfProgramHeaderTable(x2)) -> compare
```

Initially had a non-quadratic version, but...

Example: enumerations (1)

```
/* Legal values for sh_type (section type).  */
#define SHT_NULL      0 /* Section header table entry unus
#define SHT_PROGBITS  1 /* Program data */
#define SHT_SYMTAB    2 /* Symbol table */
#define SHT_STRTAB    3 /* String table */
#define SHT_RELA      4 /* Relocation entries with addends
#define SHT_HASH      5 /* Symbol hash table */
#define SHT_DYNAMIC   6 /* Dynamic linking information */
#define SHT_NOTE      7 /* Notes */
#define SHT_NOBITS    8 /* Program space with no data (bss
```

What's the "right way" to model this…

- programmatically?
- mathematically?

Example: enumerations (2)

```
enum section_type {
 NULL =     0, /* Section header table entry unused */
 PROGBITS = 1, /* Program data */
 SYMTAB =   2, /* Symbol table */
 STRTAB =   3, /* String table */
 RELA =     4, /* Relocation entries with addends */
 HASH =     5, /* Symbol hash table */
 DYNAMIC =  6, /* Dynamic linking information */
 NOTE =     7, /* Notes */
 NOBITS =   8 /* Program space with no data (bss) */
}
```

enums are a rather complex language feature…

- actually want *extensible* enums!

Example: enumerations (3)

```
let sht_null : natural = 0
let sht_progbits : natural = 1
let sht_symtab : natural = 2
let sht_strtab : natural = 3
let sht_rela : natural = 4
let sht_hash : natural = 5
let sht_dynamic : natural = 6
let sht_note : natural = 7
let sht_nobits : natural = 8
```

Some experience and observations

Performance

- "list of bytes" is a nice abstraction...
- not a good implementation
- need careful tool support

Linguistic convenience

- e.g. hex literals, fixed-width integers...
- boilerplate "for free", e.g. comparison functions

No more Mr Nice Guy

- failwith essential
- cyclic linkage relation would help (irony)
- simulating "one-pass compiler" not ideal

Conclusions & what you can do

- http://www.bitbucket.org/Peter_Sewell/linksem
- read our OOPSLA 2016 paper

Thanks for your attention!

Ask me about

- dynamic linking
- looseness problems
- dark corners
- relationship to prior work
- any other questions?

Some things we think we know

- “systems software is written in C”
- “for reasoning, we need semantics for C”
- “C compilers provide separate compilation”
- “linking is the joining of separate compiled units”

Linking: it's just how we do separate compilation of C, right?

```
$ cc -g -c -o hello.o hello.c && objdump -rdS hello.o
...
int main(int argc, char **argv)
{
   0:   48 83 ec 08             sub    $0x8,%rsp
        printf("Hello, world!\n");
   4:   bf 00 00 00 00          mov    $0x0,%edi
                        5: R_X86_64_32  .rodata.str1.1
   9:   e8 00 00 00 00          callq  e <main+0xe>
                        a: R_X86_64_PC32        puts-0x4
        return 0;
}
   e:   b8 00 00 00 00          mov    $0x0,%eax
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

Flexibility

```
/* Write formatted output to STREAM from the format string FORMAT. */
int __fprintf (FILE *stream, const char *format, ...)
{
  va_list arg;
  int done;
  va_start (arg, format);
  done = vfprintf (stream, format, arg);
  va_end (arg);
  return done;
}
ldbl_hidden_def ( __fprintf , fprintf )
ldbl_strong_alias ( __fprintf , fprintf )
/* We define the function with the real name here. But deep down in
   libio the original function _IO_fprintf is also needed. So make
   an alias. */
ldbl_weak_alias ( __fprintf , _IO_fprintf )
```

Dynamic linking

Two sides:

1. generate dynamically linkable binaries
2. actually link them

Majority of (1) already done, for overlap reasons. For (2):

- model *loading*, as done in OS or ld.so
- loading statically linked is simple enough
- dynamic linking is subtle/complex
- (ask me about dynamic linking)

Linking leakage into languages

```
if (&_IO_stdin_used != NULL)
{ /* do something ... */ }
else /* something else ... */
```

Is the **else** branch ever taken?

The knee-jerk reaction

The horror! Surely we need *a new language*.

Although:

- how do we know it covers real requirements?
- what about duplication?
- what about fragmentation?
- what about [lack of] portability?
- what about all that existing code?

Maybe in fact we need *semantics for linker-speak*.

Linker-speak (1)

For ELF targets, the `.section` directive is used like this:

```
.section name [, "flags"[, @type[, @entsize]]]
```

Linker-speak (1)

For ELF targets, the `.section` directive is used like this:

```
.section name [, "flags"[, @type[, @entsize]]]
```

Or like this (from the C compiler):

```
struct t v
 __attribute__ ((section (".data.v")))
 = { /* ... */ };
```

Linker-speak (1)

For ELF targets, the `.section` directive is used like this:

```
.section name [, "flags"[, @type[, @entsize]]]
```

Or like this (from the C compiler):

```
struct t v
 __attribute__ ((section (".data.v")))
 = { /* ... */ };
```

Or like this (living dangerously):

```
struct t unique_v
 __attribute__ ((section (".data.v,\"awG\",@progbits,v,comdat#")))
 = { /* ... */ };
```

Linker-speak (2)

```
OUTPUT_FORMAT("elf64-x86-64", "elf64-x86-64", "elf64-x86-64
OUTPUT_ARCH(i386:x86-64)
SECTIONS {
. = SEGMENT_START("text-segment", 0x400000) + SIZEOF_HEADER
.text : { *(.text) }
.hash : { *(.hash) }
.gnu.hash : { *(.gnu.hash) }
.dynsym : { *(.dynsym) }
.dynstr : { *(.dynstr) }
.interp : { *(.interp) }
. = DATA_SEGMENT_ALIGN (CONSTANT (MAXPAGESIZE),
                        CONSTANT (COMMONPAGESIZE));
.data : { *(.data) }
.bss : { *(.bss) }
.dynamic : { *(.dynamic) }
}
```

Some of the spec (3)

```
OutputSection(AlwaysOutput, Nothing, ".preinit_array",  [
    DefineSymbol(IfUsed, " __preinit_array_start ",  hidden_sym_spec)
  ; InputQuery(KeepEvenWhenGC, DefaultSort, filter_and_concat (
        fun s -> name_matches ".preinit_array" s))
  ; DefineSymbol(IfUsed, "__preinit_array_end",  hidden_sym_spec)
])
```

. . . being the AST of the following linker script fragment:

```
. preinit_array       :
{
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP ( *(. preinit_array ))
    PROVIDE_HIDDEN (__preinit_array_end = .);
}
```

Table 4.10: Relocation Types

Name	Value	Field	Calculation
R_X86_64_NONE	0	none	none
R_X86_64_64	1	*word64*	S + A
R_X86_64_PC32	2	*word32*	S + A − P
R_X86_64_GOT32	3	*word32*	G + A
R_X86_64_PLT32	4	*word32*	L + A − P
R_X86_64_COPY	5	none	none
R_X86_64_GLOB_DAT	6	*word64*	S
R_X86_64_JUMP_SLOT	7	*word64*	S
R_X86_64_RELATIVE	8	*word64*	B + A
R_X86_64_GOTPCREL	9	*word32*	G + GOT + A − P
R_X86_64_32	10	*word32*	S + A
R_X86_64_32S	11	*word32*	S + A
R_X86_64_16	12	*word16*	S + A

Some of the spec (2)

```
et  amd64_reloc r =
 match (string_of_amd64_relocation_type r)  with      (* byte width *)  (* truncate  /  sign
  |  "R_X86_64_64" ->              fun  (img, p,  rr )  ->  (8, fun (s,  a)  ->  i2n
  |  "R_X86_64_PC32" ->            fun  (img, p,  rr )  ->  (4, fun (s,  a)  ->  i2n_signed 32
  |  "R_X86_64_PLT32" ->           fun  (img, p,  rr )  ->  (4, fun (s,  a)  ->  i2n_signed 32
  |  "R_X86_64_GOTPCREL" ->        fun  (img, p,  rr )  ->  (4, fun (s,  a)  ->  i2n_signed 32
  |  "R_X86_64_32" ->              fun  (img, p,  rr )  ->  (4, fun (s,  a)  ->  i2n
  |  "R_X86_64_32S" ->             fun  (img, p,  rr )  ->  (4, fun (s,  a)  ->  i2n_signed 32
  |  "R_X86_64_GOTTPOFF" ->        fun  (img, p,  rr )  ->  (4, fun (s,  a)  ->  i2n_signed 32
(*  ...  *)
```

Some of the spec (2)

```
et  amd64_reloc r =
 match (string_of_amd64_relocation_type r)  with      (*  calculation  *)
  |  (snip)  (  (n2i  s)  + a  ))
  |  (snip)  (  (n2i  s)  + a  −  p ))
  |  (snip)  (  (n2i  (amd64_plt_slot_addr img rr  s))  + a  −  (n2i  p)  ))
  |  (snip)  (  (n2i  (amd64_got_slot_addr img rr s))  + a  −  (n2i  p)  ))
  |  (snip)  (  (n2i  s)  + a  ))
  |  (snip)  (  (n2i  s)  + a  ))
  |  (snip)  (  (n2i  (amd64_got_slot_addr img rr s))  + a  −  (n2i  p)  ))
(*  ...  *)
```

CompCert: what it does

Verify compilation as far as symbolic assembly

- then use host toolchain/runtime!

checklink checks

- that the binary contains the expected instructions
- but it also contains other stuff. . .
- . . . instructions from libc/crt
- . . . linker metadata

Linker metadata can be malicious too

computation. We introduce our design and implementation of Cobbler, a proof-of-concept toolkit capable of compiling a Turing-complete language into well-formed ELF executable metadata that get "executed" by the runtime loader (RTLD). Our proof-of-concept toolkit highlights how important it is that defenders expand their focus beyond the code and data sections of untrusted binaries, both in static analysis and in the dynamic analysis of the early runtime setup stages as well as any time the RTLD is invoked.

Shapiro, Bratus and Smith
"Weird Machines" in ELF
WOOT 2013