The Missing Link: Explaining ELF Static Linking, Semantically

Stephen Kell Dominic P. Mulligan Peter Sewell
Computer Laboratory, University of Cambridge, United Kingdom
firstname.lastname@cl.cam.ac.uk

Abstract

Beneath the surface, software usually depends on complex *linker behaviour* to work as intended. Even linking `hello_world.c` is surprisingly involved, and systems software such as `libc` and operating system kernels rely on a host of linker features. But linking is poorly understood by working programmers and has largely been neglected by language researchers.

In this paper we survey the many use-cases that linkers support and the poorly specified *linker speak* by which they are controlled: metadata in object files, command-line options, and linker-script language. We provide the first validated formalisation of a realistic executable and linkable format (ELF), and capture aspects of the Application Binary Interfaces for four mainstream platforms (AArch64, AMD64, Power64, and IA32). Using these, we develop an executable specification of static linking, covering (among other things) enough to link small C programs (we use the example of bzip2) into a correctly running executable. We provide our specification in Lem and Isabelle/HOL forms. This is the first formal specification of mainstream linking. We have used the Isabelle/HOL version to prove a sample correctness property for one case of AMD64 ABI relocation, demonstrating that the specification supports formal proof, and as a first step towards the much more ambitious goal of verified linking. Our work should enable several novel strands of research, including linker-aware verified compilation and program analysis, and better languages for controlling linking.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods; D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.4 [*Processors*]: Compilers

```
void* _int_malloc(mstate av, size_t bytes)
  { ... }
void* __libc_malloc(size_t bytes)
  { ...
    void *mem = _int_malloc(av, sz);
    ...
  }
void* __libc_calloc(size_t bytes)
  { ...
    void *mem = _int_malloc(av, sz);
    ...
  }
strong_alias (__libc_malloc, __malloc) /* These expand to */
strong_alias (__libc_malloc, malloc)   /* asm directives  */
strong_alias (__libc_calloc, __calloc) /* and/or compiler */
weak_alias (__libc_calloc, calloc)     /* attributes       */
```

Figure 1. Outline of the GNU C library's malloc and calloc

Keywords Linking, formal specification, Executable and Linkable Format (ELF), theorem-proving

1. Introduction

Programming language research focuses largely on the source-language semantics of programs and their compilation to machine code, as do the vast majority of programmers. But, beneath the surface, much real code crucially depends on complex *linker behaviour* to work as intended: as we shall see, even statically linking a small C program with a C library turns out to be an involved process and remarkably subtle. Unlike compilation, the linking process remains largely invisible and poorly studied, for programmers and researchers alike.

Consider, for example, the familiar C malloc() and calloc() functions in Fig. 1, abstracted from their GNU C library implementation. What does this code do? Looking just at the function definitions from a C-language point of view, it 'obviously' implements __libc_malloc and __libc_calloc using a third internal helper function called _int_malloc(). But this is not actually true: those two functions are not guaranteed to always call the helper as defined in this file. What happens if the user supplies their own malloc, as many programs do? C has no notion of aliases or weak definitions, but the code uses these to control behaviour in such cases: by defining malloc and calloc as alternative 'weak' names, user code may *optionally* supply its own malloc and calloc (overriding the local 'weak' alias) without affecting accesses made via the __malloc and __calloc ('strong') aliases. These strong_alias and weak_alias directives step outside the bounds of the C language: they are macro-expanded to assembler directives controlling the object file sent to the linker.

Conventional source-language semantics do not attempt to address the questions posed in the previous paragraph, but in practice linker features are used to control name binding and symbol visibility, among many other things, and many tools and libraries rely on the ability to replace or interpose on bindings. This is not a fringe issue: a significant fraction of real codebases rely on linker features to realise their intended semantics. For codebases lower in the stack, this is obvious: a typical Unix system kernel (e.g. Linux) and core libraries (e.g. glibc) make considerable use of such features. Surprisingly many user-level codebases also use a 'long tail' of linker features: it is common for mature libraries to use symbol versioning; many codebases selectively replace library functions like malloc; applications often ship wrapper scripts which play tricks with dynamic linking before executing their main binary; and so on. To say anything about the correctness of these binaries requires understanding linking in detail.

In other words, much software is not written merely in a programming language like C, but also in 'linker speak'—our term for the collection of languages by which the linker is invoked and controlled. These include the linker command-line, metadata contained within object files, assembler and compiler directives that generate that, and in fact an entire script language used exclusively by the linker. Linker-speak is currently specified haphazardly or not at all, and as a result is often used in unportable or fragile ways. Linkers are seldom well-tested, and for many purposes, the implementation is its own oracle: unlucky developers grappling with corner-case behaviour must simply work with (or around) the linker's observed behaviour. Both users and implementers lack any recourse to a reference, standard or specification. The few existing texts, most recently by Levine [24], explain the core mechanisms in detail but say relatively little about the programmer-facing features of modern linkers.

The challenge of verified toolchains makes these problems even more acute: existing approaches to verified compilation typically do not address linking at all, or address only the aspects of linking that *can* be understood in terms of a conventional source-language semantics. Previous semantic work on linking has studied it only in a highly idealised form. But truly accounting for linking must rest on accurate modelling of a large, usable subset of its features.

In this paper, we describe the first detailed, accurate formal semantics for the static linking of a realistic and widely-used executable and linkable format. Unlike previous attempts at providing a semantics for aspects of linker behaviour, our semantics is presented as an executable model testable against the observed behaviours of 'real' linkers, and not as an idealised calculus in the style of Cardelli's [5]. In particular, our executable semantics covers a large fraction of contemporary Unix linker features, and tightly models the behaviour of widely-used existing linkers on realistic link jobs—we aim to model Unix linking as it *is*, not as it (perhaps) *ought to be*. Our research contributions are as follows:

- We review and clarify the roles played by linking and loading in mainstream systems, emphasising how linkers are not merely concerned with separate compilation, but rather that 'linker-speak' has real semantic effects. Linkers complement and supplement programming languages, offering features such as memory layout control, versioning and interposition that are not exposed by programming languages.
- As a foundation for the semantics of linking on modern Unix platforms, we describe a complete, validated model of the Executable and Linkable (ELF) binary file format as it is really used—derived from the System V ABI specification [39], as extended by four platform ABIs for popular commodity microprocessor families (AARCH64, AMD64, IA32 and Power64), and additionally incorporating various extensions, notably the widely-used GNU extensions. We validate this by testing on 7 054 binaries found 'in the wild'. To our knowledge this is the first formal model of a realistic and widely-used linking and executable format, and the first formal elucidation of components of the platform ABI. It can be (and already has been) used as a front-end for other tools that need to read in ELF files, e.g. [10, 15].
- Using this model, we describe an executable specification of static linking of ELF binaries. This includes the link-time semantics for a large fraction of ELF features, together with the linker command line and the embedded script language that controls linker operation. It is sufficiently complete to statically link small C programs against a real (large) C library, covering a broad range of features.
- We demonstrate that our definitions are suitable for use in formal proof by using an Isabelle/HOL extraction of our model to prove a simple but nontrivial correctness property of AMD64 relocation: for a single instruction program we prove that the relocation machinery for a single AMD64 relocation type is correct. This correctness statement is small: it mentions only one machine instruction and one relocation type for one platform ABI. However, the proof requires reasoning about substantial chunks of the internal machinery of our linker, and has necessitated the development of a library of lemmas and definitions that can be reused in any further formal proof about our linker. Further, as a necessary stepping stone to the relocation proof, we provide termination proofs (around 1 500 lines of Isabelle/HOL source) for all functions used internally by the linker and the ELF model, obtaining a guarantee that our

linker terminates on all inputs—a result we believe is of independent technical interest. We believe that these proofs are the first formal proofs of a property related to any aspect of 'realistic' linking.

Our models of ELF files and of static linking are expressed as pure functional specifications in Lem [29], which we use to generate both executable OCaml code and theorem-prover definitions for reasoning. A working linker/link-checker can be obtained from our extracted OCaml definitions by using a thin wrapper of bespoke OCaml code to handle file I/O. The termination and relocation proofs mentioned above use a complete Isabelle/HOL version of our definitions, comprising 33 150 lines of commented Isabelle source, and these proofs demonstrate the suitability of our definitions for use in formal proof. Larger developments may (as usual) require reformulating some of the definitions for ease of proof.

Our ultimate aim is for a specification that can be used in many modes: as an actual linker producing working output; as a concise and highly readable reference implementation capturing the semantics of linking and ELF; as an input to proof-assistant mechanised reasoning about linking, and as a basis for empirical testing of linkers. More needs to be done in all these directions, but this work may already be of use to four quite disparate communities:

- developers of new or existing linkers, who may use our model as an external test oracle, both for link correctness and some aspects of ABI compliance.
- authors of certified compilers (e.g. [1, 22, 23, 40]) interested in producing ABI-compliant linkable ELF binaries, extending their source languages with elements of linker-speak, or creating a trustworthy link checker. Current efforts at checking the host linker's output, such as CompCert's cchecklink, lack a detailed model of the linker's actions, so cannot check that the linker has not inserted malicious extra content—which might even take the form of metadata rather than instructions [35].
- designers of source-level verification tools built atop the formal guarantees programming language standards and certified compilers provide. As our opening example showed, linking is able not only to refine but to override source-level semantics, so source-level reasoning alone is not sound unless augmented with knowledge of linker features—that our semantics can provide.
- researchers seeking to improve languages and toolchains, particularly for systems code: such work must somehow accommodate the various roles of linker-speak, whether by embracing it as it currently exists, or by replacing it.

Whilst much software relies crucially on linking and loading patterns more complex than static linking alone—for example dynamic linking against shared libraries, or the run-time loading of plugins—for the purposes of this paper we have restricted ourselves to specifying the semantics of static linking. Already, this is a formidable task (as evidenced by the tens of thousands of lines comprising our Lem and Isabelle/HOL specifications) yet is sufficiently self-contained to reasonably describe in a single paper. Ultimately, as we will discuss in §8, we hope to fully formalise all aspects of linker behaviour that real programs rely on to execute correctly.

Our Lem definitions and Isabelle/HOL theories are freely available from a public repository.[1]

2. Background

We focus on linking in System V-derived Unix environments. These include modern GNU/Linux and BSD variants. The *de facto* standard executable and linkable format in this environment is ELF, the Executable and Linkable Format. Microsoft Windows and Apple Mac OS (Darwin) exhibit broadly similar feature sets, each with analogous formats and mechanisms, but System V is generally a superset of both. We follow Gingell et al. [12] in referring to the Unix linker command ld as the *batch linker*.

A batch linker takes multiple modules of *relocatable object code* as input. These may be in the form of .o files, or groups of .o files bundled as 'archives' in a .a file. As output, a batch linker produces a single 'linked' binary. These binaries are usually executables and embody the input modules. The main work of the linker is to select and combine the necessary inputs, to organise them into a single logical memory image, and to concretise the symbolic references between them (encoded as metadata in the input files) into bit-patterns in the output binary (typically instruction address fields or stored pointer values).

Object files consist principally of *sections*, *symbols*, and *relocations*. Sections are chunks of bytes (code or data), treated as indivisible units by the linker: they may be moved or combined, but not broken apart. Symbols give names to particular positions within a section usually reflecting the source-level definition they represent. In an input file, a symbol may be recorded as *undefined*, meaning that the object file references it, but it does not exist in that file.

Individual references are represented by *relocation records*, sometimes called 'fixups'. These record that a range of bytes within some section must be 'fixed up' to point to the intended referent, denoted by a symbol. Before linking, these bytes hold a placeholder value generated by the compiler. After linking, they hold the encoded address of the referent. Relocations are tied closely to the encoding of instructions, and each relocation record selects an architecture-specific 'type' of relocation—a calculation for fixing up the bytes. This calculation varies according to the architecture's addressing modes, address field widths, and particular choice of pointer encodings.

A completely self-contained output binary is said to be 'statically linked'. Modern batch linkers allow some bindings to be deferred until a later *dynamic linking* step. These references are represented as relocation records and symbol

[1] https://bitbucket.org/Peter_Sewell/linksem/

metadata in the output binary, much like in the input files. The main purpose of dynamic linking is to support shared libraries which enable code to be shared between multiple processes at run time, thus saving memory. Shared library binaries themselves are also produced by the batch linker, and internally are not much different from executables.

Having composed a memory image out of its inputs, and applied relocations, the linker finally outputs a serialised form of this memory image into the output binary, such that it can be re-created by the *loader*. Concretely, any given ELF file provides one of two views: the *linkable view*, where the file is partitioned into sections as described above, and the *loadable* view, where the file is partitioned into segments. It is these segments that are copied or mapped into memory by the loader to create an executable process image. A modern Unix installation typically contains at least three loaders: the bootloader for loading the operating system, a loader in the operating system's kernel used for statically linked binaries, and a loader in user-space used for dynamic linking (ld.so).

The relationship between linker features and programming language features is complex. Many languages are said to have a 'module system', having certain features corresponding to certain linker mechanisms. Although we mostly save discussion of this for later (see §8) it is worth noting that while a typical module system provides mechanisms dealing in both abstract and concrete modules, linking is entirely concrete. Linkers have no notion of module instantiation or parameterisation; they focus only on the interconnection of concrete modules. As we will see (§3), this still encompasses a surprising variety of features, many of which have no analogue in language module systems.

The disparate and patchy specification of linking and its associated formats has posed a particular challenge for us in this work. Focusing first on the ELF format, a skeleton definition of the ELF file format is provided in the System V ABI [39], and using this specification enough structure is defined to properly parse and serialise ELF files to and from disk. However, to interpret the contents of a file more detail is needed, and architecture-specific supplements to the core specification are provided for each microprocessor family. These supplementary documents may refine, override, or fill in detail entirely missing from the core ELF specification.

Operating systems may also augment the format with their own extensions, specifying new file components that they may interpret or expect. For example, many Linux ELF binaries contain GNU extensions. These augment the standard ELF file format, detailing new structural elements that the operating system can interpret or will expect. Some of these extensions are collated in a document called the *Linux Standard Base* ('LSB') [25], though some are either documented informally in mailing list messages or not documented at all.

To capture enough of the structure of ELF files executed 'in the wild', the core specification along with microprocessor- and operating system-specific supplementary material must all be taken into account. We note here, however, that in practice there is no clean separation of the latter two sets of supplementary material. For example, the AMD64 ABI explicitly makes reference to GNU-specific extensions (see §4.2 of [27]).

Besides the ELF format, the process of linking itself is also poorly specified. In fact Unix linking has become *less* well specified over time: System V Unix's documentation included a detailed description of the linker, but this was removed when the ELF format was introduced (in revision four), leaving only a manual page. The POSIX Standard [16] even omits this, instead stating that its C and Fortran compilers 'conceptually consist of a compiler and link editor'. Although successive linkers, notably the GNU linker, offer fairly detailed manuals, these are hardly normative, and omit various details.

3. Understanding Linking

Linking is used to implement separate compilation of languages including C, C++, Fortran, Ada, Objective-C, and many others. If linking were only about separate compilation, it would have a natural specification in terms of these languages' source program semantics. Previous formal models of linking [5, 9, 13, 26, 42] have worked along these lines, seeking to provide a formal basis for separate compilation with verified safety properties. Previous research on linking in the context of verified compilation [19, 37] has done likewise.

However, linking is not simply a matter of separate compilation. Systems code and application code alike use linker features to achieve effects that are unrelated to separate compilation, which mostly cannot be expressed in the relevant source language(s), and which, in some cases, actively break the naïve semantics of the source language. In this section we survey these features.

3.1 Linker-Speak Overview

Linker-speak consists of a collection of notations, which collectively can be thought of as a separate programming or configuration language in which part of every compiled system is expressed. We survey various examples shortly; in brief, the notations break down as follows.

Arguments —these are command-line options supplied when invoking the linker. In the case of dynamic linking, environment variables serve the analogous purpose.

Scripts —most batch linkers embed a script language, used pervasively to control how sections in input files are mapped in the output file, the precise memory layout of the output file, and so on. Although programmers rarely see it, every link job is controlled by a unique 'control' script. The following is an excerpt of a simple such script.

```
SECTIONS
{
  . = 0x00001000;
  .data : { foo.o (.data); goo.o (.data); }
  .text : { foo.o (.text); goo.o (.text); }
}
```

Here, the contents of the `SECTIONS` block informs the linker how to map sections in input files to sections in the linker's output file. In particular, the linker script above instructs the linker to map the `.data` and `.text` sections in the `foo.o` and `goo.o` input files into the `.data` and `.text` sections, respectively, of the output. The dot ('.') symbol represents the current location (address) in the output file, such that the initial assignment `. = 0x00001000` sets the starting address of the `.data` section in the output file to address `0x00001000`. The `.text` section in the output is placed adjacent to the `.data` section, with the address of the `.text` section set to the size of the output's `.data` section + `0x00001000`. More sophisticated rules for mapping input sections to output can be employed; for example

```
.data : { *(.data .data.*) }
```

states that the output .data section should consist of the concatenation of all input files' sections with name .data or matching wildcard .data.*. Aside from controlling the mapping of input sections to output sections, the linker script language also provides means for controlling the program entry point, describing regions of memory and their flags, alignment, and so on, with the language including arithmetic and some forms of conditional, but no recursion or unbounded looping.[2]

The user may supply their own script, overriding the built-in default. Link jobs may also include extra 'implicit' scripts, supplied on the command line as if they were object files. These are written in a subset of the same command language, and are typically defining symbols and/or sections in a textual rather than binary notation, effectively as 'proxy' object files.

Metadata —object files contain symbols, sections, relocation records and other metadata, on which link semantics crucially depend. Many of these object file features have corresponding forms in assemblers ('directives' or pseudo-operations) and compilers (attributes), allowing the programmer to control the metadata in the assembled or compiled object file.

Our work provides a formal specification of large subsets of all of these notations. Specifically, we focus on the linker-speak of the AT&T System V linker and its descendents. In the following, any linker options or script syntax refer to that accepted by the GNU BFD linker (still the default on both GNU/Linux and BSD environments, and modelled on the AT&T System V linker).

[2] We point the curious reader to the GNU BFD linker's user manual [14], specifically Chapter 3, for a more comprehensive description of that linker's archetypal control script language.

3.2 Linker-Speak Use-Cases

Memory Placement Systems code often needs to reside at particular places in memory. For example, typical Unix kernels occupy the higher portion of the address space. This is specified by a combination of linker scripts and section name attributes. A compiler or assembler exposes a mechanism for definitions to be placed in named sections, while the linker script allows sections to be assigned to addresses, and/or to be ordered relative to one another. For example, on the PA-RISC architecture, Linux uses the following linker script to enforce a particular relative ordering of page table data. C structure layout cannot be used, because the definitions must be addressable symbolically from assembly as well as from C. In C they are declared with section attributes:

```
pmd_t pmd0[PTRS_PER_PMD] __attribute__ ((
    __section__ (".data..vm0.pmd"), aligned(PAGE_SIZE)));
```

and the sections are then placed appropriately by the linker script.

```
/* Put page table entries (swapper_pg_dir) as the first thing
 * in .bss. This will ensure 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)
        }
```

Memory placement is not expressible in source languages, but code often depends on it—for example, on some architectures, kernels recognise their own addresses by testing whether they encode a negative signed value. Control of memory placement may also be used for optimisation, to improve spatial locality [31].

Encapsulation Hiding implementation details is often achieved using linker features. Source-language encapsulation features, such as C's static modifier, may map directly to linker features, such as ELF's local symbols. However, linkers expose at least three other encapsulation facilities that are *not* supported in this way: (1) ELF symbol visibility attributes, allowing names to be scoped at the coarser granularity of dynamically-linked binaries (instead of single object files); (2) archives, where inclusion in an archive restricts an object's visibility to other modules in the link; (3) dynamic export control (`--export-dynamic`), which determines which definitions are available for binding or interposition by the dynamic linker. In addition to the linker command line, compiler options (such as gcc's `-fvisibility=hidden`) can hide more definitions by default, lessening a library's dynamic linking overheads (Drepper [8] advises using it as a matter of course). Unfortunately

-fvisibility=hidden breaks the source semantics of C++ code if it throws exceptions out of the library.[3] In general, linker features operate neither wholly above nor wholly below the language level; their use may affect both user code and language implementation internals. At present, using them correctly requires developers to understand both.

Build-Time Substitution Link-time mechanisms may be used to substitute one definition for another. ELF linking is designed expressly to allow this. For example, the semantics of archives are such that a C program may supply its own malloc.o while still linking with the remainder of the C library archive (libc.a). Indeed, it is a common performance optimisation to supply a malloc() implementation tailored to the program's allocation behaviour. Multiple definitions in .o files are generally not allowed. A linker option (such as -z muldefs) can relax this rule, causing earlier definitions (leftmost in command-line order) to take precedence over later ones. Multiple definitions are also allowed if all are marked 'weak'; an ordinary (strong) definition takes precedence, but otherwise the first weak definition is chosen. (We will see uses of weak symbols shortly, under 'optionality' and 'topology alternatives'.)

Load-Time Substitution ('overriding') Dynamic linkers offer another substitution feature: LD_PRELOAD. This environment variable can supply a named library, whose definitions take precedence over those in all other libraries (but not those in the executable). For example, it can be used to supply a new malloc at load time, which will be used ahead of the malloc in libc.so, but *not* ahead of any malloc in the executable. Using LD_PRELOAD requires assumptions about how the rest of the program is linked (e.g. here that it dynamically links against malloc()). These assumptions are invariably brittle, because in general, the split between what is statically linked and what is dynamically linked is an implementation detail of any given library. Even if the program appears to be linked dynamically against a library, say libc.so, that library's developer retains the option to link certain content statically. The GNU C library's libc.so exploits this to link its stat() implementation statically (essentially to work around possible changes to the stat structure layout). This is achieved by having libc.so be the following linker script, which acts as a proxy pulling in *both* the real shared object and an archive defining stat. (It also pulls in the dynamic linker, which supplies some definitions logically belonging to the C library.)

```
OUTPUT_FORMAT(elf64-x86-64)
GROUP ( /lib/x86_64-linux-gnu/libc.so.6
  /usr/lib/x86_64-linux-gnu/libc_nonshared.a
  AS_NEEDED ( /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 ) )
```

A side-effect is that LD_PRELOAD substitution of stat() is not possible. A similar side-effect occurs when using linker options to force early binding within libraries (-Bsymbolic)—usually motivated by the performance gained by avoiding indirection.

Interposition Interposition can be thought of as substitution where the prior definition is re-used by the substituted one—typically for instrumentation. LD_PRELOAD can be used for this too: the preloaded instrumentation can delegate to the original implementation by looking it up using dlsym(). An allocation profiler for programs dynamically linking against malloc might use this technique.[4] However, a program batch-linking its own malloc.o would require a different mechanism: the linker's --wrap option. Linking with --wrap malloc redirects malloc to __wrap_malloc, which may call __real_malloc to reach the original definition. The semantics of --wrap are subtle: it affects only references to *undefined symbols*. This means references *within the defining file* (say, a call from calloc() to malloc()) are *not* redirected. (Strictly this depends on the compiler, which is free to separate the definition from the reference, if it chooses.) In general, the user-facing semantics of features like --wrap depend on how the compiler or assembler has mapped source-level definitions and references onto linker-level sections, symbols and relocation records—itself perhaps a function of compiler/assembler options, source-level attributes, internal implementation decisions, and so on.

Optionality *Weak symbols* allow codebases to reference optional features. Unresolved weak symbols are specified to take the value 0, so the absence of a definition can be identified by a null pointer test. In practice, code like the following (from the GNU C library's freopen()) is commonplace.

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

According to the C11 standard [17], the address-of operator always returns a pointer to an object or function (§6.5.3.2 pt 3), which is necessarily distinct from the null pointer (§6.3.2.3 pt 3). However, in order to exploit weak symbols, real implementations cannot assume this.

Uniqueness and Deduplication Some programs depend on global uniqueness properties. For example, in C++, two pointers to the same function must always compare equal. When header files include inlineable functions that are address-taken, implementing this becomes difficult because the out-of-line code is repeated in multiple compiler outputs. This is solved by using linker features to deduplicate these multiple copies, ensuring a unique definition. Modern ELF versions support this using *section groups*[5]; sections are grouped by a tag string, and all but one of the group

[3] This is documented at https://gcc.gnu.org/wiki/Visibility as retrieved on 2015/11/19.

[4] Using this for malloc() is particularly tricky because a typical dlsym() implementation itself calls malloc(), setting up infinite regress.

[5] ... although it is sometimes called 'link once', after an earlier GNU extension, or 'COMDAT' after the equivalent Windows linker feature.

is discarded during the link. Link-time uniqueness is a useful mechanism, allowing the optimisation of replacing value equality (of immutable objects) with pointer equality, and is exploited by some user-level libraries [20] as well as by compilers.

Aliases In most programming languages, a definition has exactly one name. At link time, however, the same range of bytes may have multiple symbol names, each denoting the same address but with different metadata. Compilers typically expose this functionality using attributes. In our opening example, the same function has names malloc and __libc_malloc, thanks to the strong_alias macros expanding to alias("malloc"). (We will elaborate on this use of aliases shortly, in 'topology alternatives'.) Attributes often encode details crucial to a program's intended meaning, but without a rigorous specification of linker features, are prone to miscompilation. We encountered a bug in the CIL [30] C translator, which implements the alias attribute incorrectly by duplicating the function body in a separate definition, violating the intended property that all aliases have the same address. CIL cannot compile a working glibc for this reason (among others).

Topology Alternatives Aliases are often combined with substitution, visibility and optionality (weak) features to yield output objects which form different link graphs in different link contexts. The GNU C library's fprintf() implementation exhibits this (shown after preprocessing and lightly edited).

```
/* Write formatted output to STREAM from the format string FORMAT.
*/
/* VARARGS2 */
int
__fprintf (FILE *stream, const char *format, ...)
{
  /* snip */
}
extern __typeof(__fprintf) fprintf __attribute__((alias("__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 . */
extern __typeof (__fprintf) _IO_fprintf __attribute__ ((weak,
    alias ("__fprintf")));
```

The effect of the two aliases is that local code which wants to be sure of calling the local definition (perhaps because it consumes private state, or just to avoid the overhead of calling to an object further away) can use the name __fprintf. The standard name fprintf is also provided; if a substitute is provided by the user (much like the malloc substitution we considered earlier), this will not affect local calls to __fprintf. Similarly, the alias _IO_fprintf is defined for use by the libio subsystem: depending on the build, this may or may not supply its own definition, so this alias is made weak.

Introspection Linker features are used to allow programs to introspect on their own structure. The end, etext and edata symbols allow programs to test whether a pointer falls within the executable's data or text segments. This is used variously in profiling code, garbage collectors, other dynamic analyses, diagnostic pretty-printing (e.g. printing a pointer-to-data differently from a pointer-to-text) and so on. Dynamic linkers offer a richer interface in terms of dlsym() (name-to-address) and dladdr() (address-to-name) functions. Some systems code relies on the ability to introspect its own structure, often during initialisation. The GNU C library's static-linking initialisers use specially placed symbols to initialise a table at start-up. For similar reasons, the Linux kernel makes use of a GNU linker extension in which certain sections are automatically given marker symbols.

Versioning Shared libraries must allow old clients to be executed against a newer library binary. To prevent interface changes from breaking old clients, modern dynamic linkers support symbol versioning—allowing multiple versions of an interface to be exposed by a single backward-compatible binary. The linker script language and assembler pseudo-operations have extensions to support versioning, while versioning interacts with introspection: symbol names are no longer enough to identify a unique definition, so to avoid ambiguity, symbol versions must be supplied (using the dlvsym() call).

3.3 Higher-Layer Conventions

Compilers and libraries use features of the linker to achieve common ends by adopting common conventions. Specific languages often define per-platform ABI conventions to allow different implementations to interoperate. We call this *federated compilation*, as a strictly stronger requirement than separate compilation. The linker neither knows nor enforces these conventions, so a specification of linking *per se* needn't concern itself with them. However, the same arguments motivate a formal specification for them much as for linker-speak. Some specific aspects are as follows.

Code and Data Conventions ABIs specify calling conventions and some aspects of data representation typically in terms of the C language. This includes representations for integers and pointers. A C++ ABI specifies how this must be extended for C++.

Name Mangling ELF symbol names are arbitrary sequences of non-zero bytes, so can include any source-language identifier. However, most language implementations restrict symbols to the narrower set of names accepted by the assembler. C identifiers fall within this set by design, but C++ names include punctuation such as '::', which is not assembler-friendly, explaining why C++ ABIs define a name-mangling scheme.

Code Models Code models are conventions about how near or far a given definition might lie from a reference to it. Compilers allow a code model to be selected on the command line, and uses it to choose among the allowable addressing modes during instruction selection. Mismatched code models cause link errors, caused by overflow in a relocated field (a referencing instruction unable to reach its referent). Code models also define mechanisms to achieve *position independence* of shared libraries—meaning the code can run at any address without fixup, entailing it may be shared across multiple processes at different mapping addresses.[6] Although the linker is oblivious to the details of these models, it is responsible for generating certain support structures they require, based on the relocations present in the input. We will see more detail in §5.2.

3.4 Perspectives

Faced with the task of placing these complex features on a semantically firmer footing, several reactions are possible. Here we state some perspectives on this.

A Long Tail of Features Linker features can be viewed as a 'long tail': there are many, realising many disparate use cases, none is substantially more important than any other, but amounting to a considerable total. This contrasts with the usual idealised view of linking as merely a mechanism for separate compilation, where one would expect that all requirements could be met with a simple, compositional model made up of relatively few concepts. The many disparate linker features we have seen are united by (only) two properties: they make sense only when multiple modules are brought together, and putting them within one language implementation or another would be suboptimal for reasons of both duplication (if every language implementation provided its own mechanism) and fragmentation (since each would doubtless realise these features differently). It is therefore not appropriate to regard linker complexity as mere 'feature creep' to be swept away by some elegant new design; somehow, in a practical toolchain, the long tail of disparate requirements must be catered for.

Sticking with Current Designs Even accepting this long-tail property, designing new linker mechanisms and languages remains appealing, especially with the usual language researcher's aims of being more expressive, more compositional, and so on. However, the value of any new design is predicated on achieving uptake, which is a tall order given the value practitioners place on compatibility and familiarity. The alternative is to take current behaviour as a given, and formalise it as-is. Although deliberately limiting, this has the greatest potential to yield high-assurance linkers that can actually be used in practice, so is the approach we adopt. Fortunately, as we will see, the effort involved is not intractably huge.

4. The Model: ELF

Our ELF model provides types describing the concrete structure of an ELF file on disk along with more abstract types for ease of manipulation of a file's contents. Functions for parsing and blitting files to lists of bytes and for interpreting the structural elements of a file are provided. For example, functions are provided for the decoding of the section header string table, or producing a containment mapping of sections in segments. Platform-specific logic is kept separate from the main body of the formalisation via the use of higher-order functions, with ABI- or GNU-specific logic handled by function arguments. In total the ELF formalisation consists of approximately 15 300 lines of commented Lem code for core ELF, with approximately 1 800 lines for the GNU extensions and 5 300 lines for the AARCH64, AMD64, IA32 and Power64 platform ABIs.

The ELF model is currently used by `ppcmem2` [10, 15], a tool for exploring relaxed-memory behaviour on IBM POWER and ARMv8, and is used to extract an executable process image and the initial values of global variables from binaries, generating an initial machine state for the emulator.

4.1 Validation

We validated our model against a wide-range of ELF executable and linkable binaries on multiple machine architectures. Validation was conducted using two widely-deployed and mature GNU tools—`hexdump`, a tool for printing the contents of files in hexadecimal format, and `readelf`, a tool for parsing and inspecting the decoded contents of an ELF file—as trusted oracles against a set of validation binaries. On AARCH64, AMD64, IA32 and Power64 we tested against 576, 1 650, 3 222, and 1 606 binaries, respectively—7 054 in total, obtained from `/usr/bin` and `/usr/lib` on typical Linux distributions for the latter three platforms, and the contents of `/system/bin` and `/system/lib` of an Android 5.1.1 smartphone for ARM.

Using our model we wrote a tool that emulated a subset of `readelf`'s functionality. Using an automated `diff` tool, the output of the real `readelf` was compared with the output of our tool on the validation binaries, testing the parsing and decoding of the file-header, section header table, program header table, dynamic section, relocation sections, and symbol and string tables.

Using `hexdump`, a 'roundtripping' property of the parsing and blitting functions was also validated, ensuring that parsing and then immediately blitting a binary preserved byte-for-byte compatibility with the original file. This requires the tracking of 'dead' data in between the structural elements of a file to ensure that the byte-for-byte condition holds.

Validating the model against binaries found 'in the wild' on real machines revealed many sources of incompleteness in the various source specification documents. For example, constructions relating to the ELF prelinker, such as the dynamic section type `DT_GNU_PRELINKED` are found in a large number of deployed ELF binaries, yet are not mentioned in any specification document, being mentioned only in passing in mailing list messages (e.g. 'prelinking' features [18]), and prior to validation were unknown to us.

[6] It says nothing about whether the code's *semantics* depend on its load address; position-independent code is perfectly able to introspect its own load address, to branch on pointers, etc..

One of our aims with this work is to create a comprehensive, validated set of definitions suitable for software verification purposes, serving as a foundation for further work in the area. Toward this end we have extracted Isabelle/HOL theory files from our Lem source model and provided hand-written termination proofs for recursive functions in the ELF model and in the linker that sits atop this model, gaining a guarantee that our linker terminates on all inputs (the linker is described in the next section, §5). This Isabelle/HOL code has been used in an experiment with formal proof, described in §6.

Whilst we prefer to use Isabelle/HOL, we are aware that the formal methods community has not come to a consensus on a single theorem proving environment to work with, and other researchers may wish to use Coq, or HOL4, or another system, when working with our definitions. HOL4 and Coq theories, extracted from our Lem model, will be made available in due course.

We started experimenting with Isabelle/HOL extractions of our Lem model quite early in the development process, and our use of formal proof was a key component in the validation process of the ELF model. In particular, many low-level bugs in the ELF formalisation were discovered not by the testing-based validation process described above, but by failed proof attempts. For example, an incorrect byte-ordering in the parsing functions for eight-byte types used by the ELF format was discovered due to a failed proof attempt when using an early version of our Isabelle/HOL extraction.

5. The Model: Linking

Building on this formalisation of ELF, we can now formalise the operation of a linker and relevant linker-speak features. We focus on static linking of executables in this paper.

5.1 Overview

Our formalisation takes the form of an executable specification that can operate as both a linker and a link checker. It is designed around the abstraction of *memory images* and associated *annotation* metadata. Linking is expressed as a transformation of memory images. Each input ELF file is represented abstractly as a partial memory image, consisting of a collection of *elements*, mostly mirroring ELF sections. Byte ranges within elements are labelled with metadata *tags*, mostly mirroring ELF metadata such as symbols, relocations, and section properties. At the end of the linking process, a single memory image is assembled, which is transformed back into an ELF file.

The specification is invoked the same way as the GNU BFD linker, supporting the same command-line options. To use it as a link checker, it is run with a pre-existing output file (named with `-o`, as usual), whose memory image is checked against the one produced by the specification. Incidental details of the input ELF file, such as the ordering of ELF symbols or section headers, are not significant; only the memory images, in terms of their contents and addresses, are compared.

Checking is complicated by the surprising amount of non-deterministic choice, or 'looseness', available to the linker. The very simplest link jobs are entirely deterministic, but many linker features introduce opportunities for per-linker or per-run variation. At present, the variation must be captured explicitly in the specification as 'personality functions': the core specification is factored so that each kind of non-determinism is resolved by a separate function, allowing emulation of different linkers. Linker personalities are complex. At present, our specification includes a single personality, based on the GNU BFD and gold linkers, but we have uncovered certain bugs and complications which prevent it from precisely emulating either one. For example, even with optimisations and relaxations disabled, the GNU linker sometimes rewrites instructions in the input binaries—a divergence our memory-image check (correctly) flags up. For this reason, realistically-sized link jobs (e.g. those including a C library) currently do not pass the checker. Modelling the GNU linker's optimisations in greater detail, including allowing personality functions to rewrite instructions, would address this. We discuss further 'loose' aspects of linker behaviour in §5.3.

The notion of memory images generalises to *symbolic* memory images, in which each memory image element's address and content may be expressed in terms of symbolic variables and unordered concatenations of fragments, rather than precise addresses and bytes. This approach potentially allows a family of possible links to proceed at once, accommodating the looseness inherent in checking for 'any valid link' without the need for a precise specification of personality. Designing such a symbolic representation, and finding efficient ways to test its satisfiability, is a complex problem which our ongoing work is addressing.

5.2 Linking `bzip2`

In the C programming language, a simple program such as 'hello, world!' exercises very few features of the language, and can be compiled even by a toy compiler. However, for a linker, even the smallest C program amounts to a complex job, since it links with the C library—one of the most complex libraries on the system, in terms of the linker features it exercises. Our model can cope with the large link jobs that arise from linking small but real C programs, not limited to hello-world. In this section we outline what happens when such a C program, bzip2, compiled with gcc for x86-64, is linked against uClibc[7], a fully-featured C library slightly simpler than the standard GNU C library. The reason for using uClibc is to avoid certain GNU extensions, namely IRELATIVE relocations and IFUNC symbols, for which support remains in the bug-fix stage at the time of writing. Our formalisation captures (executably) the following steps.

[7] `http://uclibc.org/`

```
let command_line_table = [
  (["-o"; "--output"],          (["FILE"],  []),  fun args -> set_or_replace_opt (OutputFilename(head (fst args))), "Set output file name");
  (["-Bsymbolic-functions"],  ([],  []),          fun args -> set_or_replace_opt (BindFunctionsEarly), "Bind global function references locally ");
  (["-Ttext-segment"],         (["ADDR"], []),  fun args -> set_or_replace_opt (TextStart(parse_addr (head (fst args)))), "Text segment address");
  (["-("; "--start-group"],  ([],  []),          fun _ -> (fun state -> start_group state), "Start a group");
  (["-)"; "--end-group"],    ([],  []),          fun _ -> (fun state -> end_group state), "End a group");
  (* ... *) ]
```

Figure 2. Excerpt from the specification of GNU linker command-line options

Parse Command Line This stage is responsible for identifying input files and link options. Our link command, slightly simplified to omit directory names and library path lookups, is as follows.

```
ld -m elf_x86_64 -static -o bzip2 crt1.o crti.o crtbeginT.o \
 blocksort.o bzip2.o bzlib.o compress.o crctable.o \
 decompress.o huffman.o randtable.o \
 libm.a -( libgcc.a libgcc_eh.a libc.a -) crtend.o crtn.o
```

Only the .o files on the middle two lines came from compiling the program; those above and below are supplied by the compiler (libgcc.a, crt{1,i,n}.o) and C library (libc.a, crt{beginT,end}.o). Other options are modifiers; some apply to the whole link (like `-m elf_x86_64`, selecting x86-64 output) while the rest affect only the input files that follow them, or until negated: here `-(` is negated by `-)`.[8] These bracket options set up 'groups' of archives, affecting symbol resolution semantics: groups permit cyclic references among archives, while normally archives can only be referenced by objects appearing to their right. Some options may be meaningfully repeated (such as `--defsym name=expr`, which defines a new symbol). The command line is formalised as an interpreter, whose state is the collection of input files and currently active modifiers. A list of option definitions defines the next-state function: this list (see Fig. 2) resembles the linker's `--help` text, but supplies each option's semantics as a function from state to state. Complex options such as `--push-state` exist, requiring that a state include both a current value and a stack of previously saved values.

Resolve Symbols to Objects Although 'only' 17 files appear in the command, the four archives contain a total of 1 095 objects, in addition to the 13 objects named directly. To discard those that are unneeded, the linker next resolves symbol references between all 1 108 objects. The semantics of symbol resolution are complex, as we noted regarding the treatment of archives and groups. Our semantics is factored into an 'eligibility predicate' answering whether a given reference can bind to a given definition, and an ordering on eligible definitions such that the first eligible definition is the intended referent. The predicate is shown (slightly abbreviated) in Fig. 3. The ordering is based on command-line order, but also accounts for the semantics of substitution: definitions in relocatable files take precedence over archives,

```
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
        (* Weak references *can* be resolved to archive members...
         * if the reference itself is also in the archive. *)
        ((not ref_is_weak) || ref_and_def_are_in_same_archive)
        && (
             ref_is_leftmore
          || ref_and_def_are_in_same_archive
          || ref_is_in_group_with_def
        )
      else true
in let eligible_defs = List.filter def_is_eligible
    defs_and_linkables_with_matching_name
in (* ... *)
```

Figure 3. Excerpt from the eligibility predicate used to form symbol bindings

hence providing the semantics necessary for the malloc.o substitution example (§3.2). Once all symbol references are resolved, any unreferenced objects can be excluded. In our case, this leaves 141 objects in the link.

Generate Support Structures The linker must generate support structures used by certain code models and relocation schemes. In most ABIs, these include the GOT (global offset table; a table of pointers) and PLT (procedure linkage table; a table of trampolines). These are used for indirect addressing, when code compiled with narrow addressing modes must reach definitions located far away. The linker generates a GOT consisting roughly of one entry for each distinct symbol definition used in a GOT-based relocation. Support structures are particularly critical for dynamic linking, to allow libraries to be loaded across the entire address space and to support lazy binding, but static link jobs may also require generation of a GOT (frequently) and a PLT (occasionally). Although support structures are a function of the overall link contents, they must be generated early, *before* the linker script runs, to give the linker script control over their placement, hence before any output memory image exists. This requires ad-hoc modelling; for example, the GNU linker pretends that these structures reside in the first input object. To link bzip2, the GOT is necessary mainly to sup-

[8] Confusingly, `-static` is also of this kind: if any `-lX` options preceded it, the libraries they denote might be linked dynamically, meaning the output would *not* be a (fully) statically linked binary.

port relocation schemes for thread-local storage (required for the thread-local errno); no PLT is necessary (unless linking with glibc, whose IFUNC symbols do necessitate a PLT). We must also support TLS relocations (below).

Optimise Relocations and Instructions Immediately before generating these support structures, many linkers apply optimisations to the relocations and, in some cases, to the instructions that use them. For example, in our bzip2 link, to avoid use of the GOT when static-linking, the GNU linker will turn the following mov, which loads an address from the GOT

```
48 8b 05 00 00 00 00      mov    0x0(%rip),%rax
        ---------- to be relocated:
                   R_X86_64_GOTPCREL __libc_stack_end-0x4
```

into the following lea which calculates it directly.

```
48 8d 05 3d 1e 20 00      lea    0x201e3d(%rip),%rax
        ---------- applied relocation:
                   R_X86_64_PC32     __libc_stack_end-0x4
```

Some ABI documents list 'standard' optimisations that may be applied. A linker is free to use them or not.[9] Our model currently lacks knowledge of the instruction set architecture, so does not capture these optimisations. Since the GNU linker does not currently provide any way to disable these optimisations (even when supplying command-line options intended to disable optimisations), we diverge from it here: our GOT will contain more slots, subsequent address assignments will be skewed, and more instructions will indirect via the GOT.

Compose Output Sections The main pass over the linker control script assigns input sections to output sections. The default GNU linker script is 226 lines long. Our formalisation defines the linker script language's abstract syntax in Lem, using Lem for arithmetic and pattern-matching logic. We then manually translated the default script into an AST value in Lem. For example, a fragment of the original linker script

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

is represented as the following.

```
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)
])
```

Section composition is mostly concatenation, but ELF section flags can mark sections as mergeable. Most commonly these are sections containing strings, signified by an additional section flag. Again, it is up to the linker whether or not sections are merged. Symbol definitions made in the script can *substitute* (§3.2) for definitions in the input files. This means that the reachability calculation used to discard unwanted inputs was not definitive: it may have included some objects that are no longer needed, since the relevant bindings were altered during script execution.

Garbage Collection To provide a finer-grained removal of unwanted input, and to compensate for the problem that, as we noted, the initial reachability calculation is subject to invalidation, the command line may request an additional garbage collection pass (`--gc-sections`). This also entails delaying address assignment, to avoid allocating addresses for sections that will be collected. Normally `--gc-sections` is not used, and we currently do not model it.

Assign Addresses to Symbols Another pass over the linker script now assigns addresses to output sections. Addresses can be computed explicitly in the script using arithmetic, and can depend on the size and address of any section placed earlier in the script. By default, addresses are computed using a location counter that is automatically incremented, rounded up to account for section alignments specified in input files.

Apply Relocations Once addresses have been assigned, relocations can be applied. This is actually where linking happens. Our bzip2 example requires 1 941 relocations of six different kinds: 32, 32S, GOTPCREL, PC32, PLT32, GOTTPOFF. The last of these is a thread-local relocation, supplying the relocated instruction not with an address but with an *offset* in the thread-local array.

Generate Output Finally, we have a relocated memory image. We can compare this against the input bzip2 object; this comparison fails because the GNU linker has altered some instructions (and generated a smaller-than-expected GOT). Our image nevertheless otherwise corresponds very closely to the GNU linker output; it can be serialised straightforwardly into an output ELF file and executed with identical behaviour (including passing test cases).

5.3 Looseness

Linking is deterministic in simple cases. For example, a link job controlled by a known linker script and whose input consists only of simple freestanding object files will produce a deterministic memory image—unless it contains common symbols, orphan sections, section groups, mergeable sections, or if the linker must insert padding or generate nontrivial support structures. Nearly all real link jobs have some of these properties. Here we summarise these sources of looseness.

Output Ordering When concatenating a collection of .text sections, say, the sections must generally be ordered by the order of the originating objects on the command line. However, archive members are ordered in (to quote the GNU ld manual) 'the order in which they are seen during the link', a

[9] ... the newer gold linker does not currently apply them.

```
let amd64_reloc r =
  match (string_of_amd64_relocation_type r) with   (* byte width *) (* truncate / sign-ext *) (* calculation  *)
  | "R_X86_64_64" ->          fun (img, p,  rr) -> (8, fun (s, a) -> i2n            ( (n2i s) + a ))
  | "R_X86_64_PC32" ->        fun (img, p,  rr) -> (4, fun (s, a) -> i2n_signed 32 ( (n2i s) + a - p ))
  | "R_X86_64_PLT32" ->       fun (img, p,  rr) -> (4, fun (s, a) -> i2n_signed 32 ( (n2i (amd64_plt_slot_addr img rr s)) + a - (n2i p) ))
  | "R_X86_64_GOTPCREL" -> fun (img, p, rr) -> (4, fun (s, a) -> i2n_signed 32 ( (n2i (amd64_got_slot_addr img rr s)) + a - (n2i p) ))
  | "R_X86_64_32" ->          fun (img, p,  rr) -> (4, fun (s, a) -> i2n            ( (n2i s) + a ))
  | "R_X86_64_32S" ->         fun (img, p,  rr) -> (4, fun (s, a) -> i2n_signed 32 ( (n2i s) + a ))
  | "R_X86_64_GOTTPOFF" -> fun (img, p, rr) -> (4, fun (s, a) -> i2n_signed 32 ( (n2i (amd64_got_slot_addr img rr s)) + a - (n2i p) ))
 (* ... *)
```

Figure 4. Excerpt (slightly simplified) from the specification of x86-64 relocations, used in linking our bzip2 example. The parameters p, s and a denote respectively (as in the ABI specification) the relocation site address, symbol address and addend.

detail of the linker's dependency graph traversal algorithm. As a result, archive members may appear in different orders. The same ordering non-determinism applies to common symbols.

Padding Lengths and Values When padding sections to satisfy alignment constraints, both the amount and the contents are not fully determined. Superfluous padding is never desirable, so arguably a bug; we filed a bug on the gold linker[10] inserting too much padding and of the wrong byte values. In practice, linkers use zeroes to pad data sections, and nop-sequences to pad code. The latter are essential to allow control to flow between abutting sections even in the presence of padding. This is sometimes done (e.g. the GNU C library splits some code between crti.o and crtn.o, such that control flows across the join). For an *n*-byte nop, many choices of instruction may be available, depending on the architecture.

Relocation, Optimisation, Merging, Section Groups As covered in §5.2, linkers are free to optimise certain relocations, sometimes replacing instructions. They are also free to merge mergeable sections, or not. Sections that are members of section groups (§3.2) compose differently from ordinary sections: all but one section in the group is discarded, but the choice of which to discard is left to the linker.

Segment Padding At boundaries between segments (a.k.a. memory mappings), the linker's address assignment algorithm faces trade-offs about disk space (zeroes in the output file) and memory (wasted space in mapped pages). The GNU linker script language's ALIGN_DATA_SEGMENT feature inserts an amount of padding calculated to optimise this trade-off. Our specification revealed an inconsistency between the GNU linker's manual and behaviour.[11]

Orphan Section Placement Sections not matched by any clause in the linker script are still included in the output. They can be placed in any output section having suitable flags; the choice is left to the linker.

[10] GNU binutils bug #18979, at https://sourceware.org/bugzilla/show_bug.cgi?id=18979 as retrieved on 2016/8/24.

[11] GNU binutils bug #19203 (at https://sourceware.org/bugzilla/show_bug.cgi?id=19203 as retrieved on 2016/8/24)—now fixed

Linker-Generated Structures The GOT, PLT and other run-time structures (§5.2) are effectively lists, whose order is arbitrary. In practice, the order adopted often reflects the linker-internal hash table implementation.

Relaxation A family of linker optimisations known as 'relaxations' can rewrite content at relocation site (choosing a shorter calling sequence, say) and section boundaries (overlapping leading and trailing padding in exception handling information, say). These are mostly specific to the instruction set; as before, although conceptually a linker need not understand instruction encodings, most do.

Phase Anomalies A linker necessarily makes multiple passes over its inputs. Passes include enumerating inputs, calculating output section layout, calculating addresses, applying relocations, and so on. Some linker features interact in ways which induce circular dependencies between these passes, which the linker resolves in arbitrary and undocumented ways. One example is input enumeration: the linker 'pulls in' archive members to provide symbols required by other input objects. However, the linker script might subsequently provide its own definition for some symbol, obviating the need for a definition pulled in. Whether such obviated inputs are removed from the link is a phase-order detail; in our experience they are not. Contents of GOTs and other support structures also tend to reveal these phase details (since the GOT must be sized before linker scripts are fully evaluated, hence before symbol bindings are finalised).

5.4 Status

The above discussion has highlighted several ways in which the current model diverges from existing linkers; we briefly summarise these here.

Partial Specifications Many aspects of the specification are only 'filled in' sufficiently for our small use cases (helloworld, bzip2, etc.). For example, many command-line options are not specified, nor are some of the more obscure relocation kinds. Adding these is an incremental effort, since the necessary support structures are already in place (abstract notions of elaborated command-line, relocation support structures, and so on). Certain ELF and linker script features are also not currently implemented for similar reasons; these include section groups (a feature used heavily by

C++ link jobs), various sorting and discarding behaviours selectable within linker scripts, and so on. Again, adding each of these is expected to be a localised task.

Linker Script Parser For link jobs that require a linker script other than the default linker script supplied by our linker, such as most operating system kernels, adding a linker script parser is necessary. This is not technically demanding, since there is a close correspondence between our Lem datatype and the actual linker script syntax.

Loose Comparison For very simple link jobs that do not exercise sources of looseness, it is already possible to test our specification against real linkers and expect to find an exact match in their output memory images. However, given the many sources of looseness identified in the previous section, testing on realistically sized link jobs requires adding a loose 'symbolic' comparison function. The notion of memory images is already a reasonable basis for this, but, as noted earlier, requires some further work to deliver a tool that can accurately handle looseness in large link jobs (§5.1).

Instruction Optimisation A large obstacle to using the default GNU BFD linker is the support for instruction-level optimisations illustrated earlier. Since these are prescribed by the ABI specification, creating a series of pattern-matching bytewise translations is a feasible approach, and does not require a full-scale integration of instruction set semantics.

Dynamic Linking Dynamic linking is ubiquitous on commodity systems, and our specification is designed to support it. Note however that dynamic linking involves two steps: generating individual dynamic objects as output (by ld), and actually doing the load-time link of many such objects (by ld.so). The first is a minor extension to our specification, and as noted, existing specifications of GOT and PLT generation are a useful foundation. The second requires considerable additional front-end logic (locating participating binaries, building a link map, assigning load addresses, etc.), effectively amounting to a separate specification for *loading*. For static linking, loading is very straightforward, since the file embodies the entire memory image; our ELF specification already includes this. For dynamic linking, the ld.so loading process needs its own additional specification.

Performance, Housekeeping etc. The current specification codebase has various defects as would be expected for an undertaking of this scale. The code contains various workarounds for bugs in Lem, for which proper fixes are anticipated. The generated OCaml code suffers performance problems owing to the data structures fixed by Lem libraries; large link jobs take a long time to complete. This does not affect the functional correctness of our linker specification, nor does it require changes to the specification *per se*, but affects its usability at present. The code is also marked in various places with notes of uncertainty about what the 'correct' specification is, in need of clarification by correlating both experimental observations and documentation; resolving these is an ongoing task. Like any large software artefact, there are a sprinkling of code quality issues and known minor bugs. Finally, there are missing bindings for certain definitions used when extracting to backends other than Isabelle/HOL and OCaml.

6. Formal Proof

As an initial step in using our definitions for formal proof we have proved a simple but nontrivial correctness property of the relocation process, a central mechanism involved in linking. We use an Isabelle/HOL extraction of Fox's x86-64 model [11], relying on Fox's specification of x86-64 instruction encoding, in conjunction with our own Isabelle/HOL definitions extracted from the Lem model in the proof.

We fix two single-instruction programs, both consisting of an x86-64 unconditional `MOV` instruction that loads the immediate constant 5 to an absolute address in memory, say to a C global variable. This address is not fixed, as we quantify over the absolute target of the move instruction. The first of the two programs will have the target address of the move supplied by our linker's relocation mechanism, and we therefore initially set the target address to zero. The second of the two programs will remain untouched by relocation and has the quantified address as the target address of the move.

We construct a relocation entry describing a relocation that will be applied by the linker to the first of our two programs described above, and define two sections: a `.text` section that will contain our encoded program, and a `.data` section which we will use as scratch space for writing to with our `MOV` instruction. Both sections have fixed start addresses and fixed sizes. We create a symbol, arbitrarily called `test`, whose reference is within the bounds of the `.text` section and whose definition is within the bounds of the `.data` section. A symbol reference denotes a location where relocation will take place; here it coincides with the memory address of the address field within the encoded instruction in the `.text` section. A symbol's definition denotes where the relocated address field will point to after relocation; here this is an address within the bounds of the `.data` section.

Call an address *valid* when it lies within the bounds of the `.data` section, and call a finite map from 64-bit words to 8-bit bytes a *flat memory*. We prove that for any valid address a, a 'correct' flat memory can be obtained from the memory image produced by the linker's relocation machinery after relocating `test`. Here, a flat memory is 'correct' when (1) it is pointwise equal to the flat memory obtained from encoding our fixed instruction with address field set to a, and (2) the base address at which the encoded bytes reside equals the start address of the `.text` section. Note that the theorem statement is parametric in the address at which the symbol's definition resides. This means that simply executing the linker's relocation machinery is not sufficient for establishing that the statement holds; the use of proof if necessary.

This theorem is about relocation, and not about linking as a whole. Nevertheless, the theorem is nontrivial: it involves reasoning about the linker's relocation and symbol resolution machinery, about aspects of the AMD64 ABI, and about internal data structures used within the linker. Further, we believe that this theorem is interesting for two reasons. First, though the correct execution of much real-world software depends crucially on relocation being applied correctly, relocation itself is hard to validate adequately by testing due to each ABI supplying a unique set of relocation types, necessitating a large number of test cases to guarantee full coverage. This problem becomes more profound as the number of relocation types supplied by an ABI grows. Though the AMD64 ABI specifies around 20 different relocation types, the ARM64 and Power64 ABIs supply around 120 and 80 custom relocation types, respectively, causing a combinatorial explosion in the size of test suite needed to ensure adequate coverage. Formal proof therefore can be profitably used to achieve a level of certainty that relocation is being correctly applied that is hard to achieve by any other means. Second, we believe that this theorem is the first formal proof of any property related to relocation, and the first formal proof of any property related to linking for realistic executable and linkable formats.

The proof of the statement discussed above does not come close to demonstrating the total correctness of our linker, nor even the linker's relocation machinery. However, the theorem serves to demonstrate that our definitions can be used for formal proof, given how sensitive to the particular form of definitions that activity often is, and we see this theorem merely as a stepping stone to a wider application of formal proof in this area. In particular, we see no impediment to extending our theorem detailed above to all relocation types supplied by the AMD64 ABI, all instructions supplied by the x86-64 instruction set, and to an arbitrarily large sequence of instructions, which we believe would establish the total correctness of our linker's relocation machinery for the AMD64 ABI. Further properties that could be proved about our linker are discussed in §8.

A significant portion of the total effort spent trying to prove our theorem was spent proving generic lemmas about the internal data structures, ordering functions, and so on, used within the linker. These can be reused. In total, the proof of the theorem consisted of around 4 500 lines of tactic-driven Isabelle/HOL proof script. Of this total, around 1 500 lines of proof script were dedicated to manual proofs of termination for all recursive functions, and around 2 500 lines of proof script were dedicated to generic lemmas about the internal data structures of the linker. All of these lemmas can be reused in any further verification project, with around 500 lines of proof script being specific to the proof at hand.

7. Related Work

We believe we are the first to provide a comprehensive, validated formalisation of a realistic linking and executable file format, the first to formalise large parts of the Application Binary Interface of common commodity platforms, the first to build an executable specification of 'realistic' linking, and the first to formally prove any property related to any aspect of 'realistic' linking. Whilst Kennedy et al. [21] are able to generate Microsoft Portable Executable (PE) files from a Coq formalisation of X86, they formalise only enough of the PE format to obtain a working executable from machine code.

Previous theoretical work has addressed some limited aspects of linking. Cardelli's work on program fragments and modularisation [5] primarily viewed linking as a way of facilitating the separate compilation of modular programs (see Cardelli's Theorem 7.3, for example), and formalised linking via the use of linksets. This approach, focusing on separate compilation, was followed by considerable further theoretical work [9, 13, 26, 42]. However, as we stress in this paper, the linker has many roles over and above that of separate compilation, that these works do not address.

Closely related to the theoretical work above is work on module systems. As we noted earlier in §2, linking contrasts with the module systems of many programming languages by being almost entirely concrete, having no notion of module parameterisation, nor of module instantiation. The linker is responsible only for creating the concrete 'wiring', or binding topology, of the output binary, but not for instantiating abstract modules, with the programmer having little direct control of which module gets wired to which other. Rather, bindings are formed entirely according to symbol names, without regard to the name of the module (file) supplying or requiring them. This makes a linker's role in 'programming in the large' comparable to a (somewhat inexpressive) module interconnection language [7]. Subsequent work in this area has extended Unix-style batch linking with explicit hierarchy [33] and dynamic linking with greater runtime interposability [34]. Linkers' substitution and interposition features (see §3.2) correspond to features in certain 'mixin'-style module systems [3, 4]; in particular, dynamic linking has been shown to relate closely to mixin layer composition [36].

Verified compilation projects have also touched on aspects of linking. The CompCert compiler generates assembly code which is assembled and linked via the host toolchain. Rather than formalise linking directly, a *post-hoc* tool [38, Section 7] checks that the output binary reproduces the expected reference graph. The check is necessarily very partial: the host toolchain requires extra code to be linked in for its operation (e.g. C library startup code), which the checker must trust. This approach is also limited to separate compilation of a single language, and does not apply to realistic existing codebases using languages besides C, including linker-speak.

Compositional CompCert [37] extends the compiler and proofs to handle the separate compilation of modules, obtaining the first verified separate compiler for C, but required significant changes to the compiler's proof of correctness. The scale of these modifications motivated the more lightweight approach taken by Kang et al. [19]. This work limited the code being linked to that produced by CompCert, and therefore required far fewer changes to the CompCert proofs. Like the theoretical work mentioned earlier, these extensions still view linking primarily as a means of achieving separate compilation, and ignore the many other roles of the linker.

Wang et al. [41] verified a compiler for the Cito programming language in Coq. One novelty of this compiler was the ability to link compiled Cito programs against assembly code produced by other compilers, and reason about the linked code with a Hoare-style program logic. Again, this work only considers linking as a vehicle to enabling separate compilation, albeit in a mixed-language setting.

Other verified compilers have tended to ignore linking, with the exception of the Piton project [28] which included a simple link-assembler for a low-level assembly-like language. The CerCo compiler [1] is limited to single-module programs, as are the C0 compiler [32], Chlipala's compiler [6], and the CakeML compiler [22] to name some notable recent projects; the latter uses the host toolchain to link in a small amount of native code to implement string input-output routines.

8. Future Work

By bringing linking into the realm of mechanised semantics, we hope to provide a concrete foundation for several strands of new research.

Specification of Higher-Level ABI Conventions Federated compilation relies on ABI conventions above the linker level, include calling conventions and data layout (§3.3). Formal specification of these, although separable from linking, is essential to any full specification of a toolchain.

Relating Source Languages to Linker Abstractions Language implementers need ways to state their assumptions and guarantees about the link-time environment and its mappings to and from the source language. This could then enable accurate source-level reasoning about linker-supplied definitions (like the introspective `end` and `etext` symbols, §3.2), or linker-invoked features such as visibility attributes, aliases, and so on.

Program Analyses Accounting for Linker-Speak Program analysis of real codebases involving linker-speak must account for its semantics. Several approaches are possible: link-time reasoning near the machine level, perhaps following Balakrishnan and Reps [2] in seeking visibility of errors introduced during compilation, or perhaps performing intermediate-language reasoning in a linking-aware context, analogous to current approaches to link-time optimisation in LLVM and gcc.

User-Facing Improvements Much of the user-facing complexity of linking can be argued as unnecessary. Linker-speak is not a well-designed language: its incantations often implement simple properties using low-level mechanisms. Linker behaviour (particularly errors) have a habit of mystifying programmers, even experienced language implementers. (Two anecdotes in this space include GHC bug 8935, in which GHC developers initially mistook the completely standard semantics of `dlsym()` for a bug, or OCaml Mantis issue 6462 which incorrectly blames a program corruption bug on a claimed lack of support in the linker.) Weighty documents of intricate user advice, like Drepper's [8], suggest a need instead for higher-level policy-like abstractions, from which a smart toolchain can figure out how to perform the link. We would also like ways to factor linker-speak so as to avoid the potential for link-time interference between user-supplied and toolchain-required link behaviour (as with our `-fvisibility` example, §3.2).

Further Work on Formalising Separate Compilation Several recent strands of work in the field of verified compilation have touched upon verified separate compilation (see §7). Though we have been careful to point out that 'real' linkers provide many services to programmers, separate compilation remains a major use-case for linkers, and we aim to use our linker to study the source-level semantics of programming languages with mechanisms for separate compilation, and their verified compilation.

Extending and Enhancing Existing Verified Compilers Our models are a potential means of enhancing the 'trust story' of existing verified compilers by eliminating the dependency on the untrusted host toolchain, and any *post hoc* link validation tools, when producing binaries. We aim to extend the CakeML compiler to produce ABI-compliant binaries directly, which currently produces executable binaries by wrapping generated machine code in a thin layer of hand-written C, and relying on the host platform toolchain to generate the final binary.

Further Verification We plan to extend the work detailed in §6, where our proof of concept proved only the commutation of a single relocation type with a single, fixed machine instruction, for a single platform ABI. This theorem is not sufficient to establish the total correctness of our linker's relocation machinery. However, extending this to all relocation types supplied by a given ABI against arbitrary numbers of machine instructions supplied by an instruction set would, we believe, establish the total correctness of relocation for a given ABI.

Our Isabelle/HOL termination proofs have established that our linker terminates on all inputs. It would be interesting to try to identify some predicate on ELF files such that if a set of files all satisfy the predicate then our linker will terminate in a non-failing state when linking them.

In the longer term we aim to establish the total correctness of our linker. This is an ambitious goal. What total correctness for a 'realistic' linker—which as we have argued in this paper provides many services to users over and above the facilitation of separate compilation—means is not *a priori* clear and arguably a research contribution in its own right.

Acknowledgments

We acknowledge funding from EPSRC grants EP/H005633 (Leadership Fellowship, Sewell) and EP/K008528 (REMS: Rigorous Engineering for Mainstream Systems Programme Grant).

References

[1] R. M. Amadio, N. Ayache, F. Bobot, J. B. Boender, B. Campbell, I. Garnier, A. Madet, J. McKinna, D. P. Mulligan, M. Piccolo, R. Pollack, Y. Régis-Gianas, C. Sacerdoti Coen, I. Stark, and P. Tranquilli. Certified complexity (CerCo). In *Proceedings of the 3rd International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA)*, pages 1–18, 2014.

[2] G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32:1–84, 2010.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming and Object-Oriented Programming Systems, Languages, and Applications (ECOOP/OOPSLA)*, pages 303–311, 1990.

[4] H. Cannon. Flavors: a non-hierachical approach to object-oriented programming. Technical report, Symbolics Inc., 1982.

[5] L. Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 266–277, 1997.

[6] A. Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 93–106, 2010.

[7] F. DeRemer and H. Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software*, pages 114–121, 1975.

[8] U. Drepper. How to write shared libraries, December 2011. Available at `http://www.akkadia.org/drepper/dsohowto.pdf`, retrieved 2015/11/19.

[9] S. Fagorzi and E. Zucca. A calculus of open modules: call-by-need strategy and confluence. *Mathematical Structures in Computer Science*, 17:675–751, 2007.

[10] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016.

[11] A. C. J. Fox. Improved tool support for machine-code decompilation in HOL4. In *Proceedings of the 6th International Conference on Interactive Theorem Proving (ITP)*, pages 187–202, 2015.

[12] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. Shared libraries in SunOS. In *Proceedings of the USENIX Summer Conference*, pages 375–390, 1987.

[13] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 250–261, 1999.

[14] GNU Project. ld, the GNU linker. Available at `https://sourceware.org/binutils/docs/ld`, retrieved on 2016/8/24.

[15] K. Gray, G. Kerneis, D. P. Mulligan, C. Pulte, S. Sarkar, and P. Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 48)*, pages 635–646, 2015.

[16] IEEE POSIX Standard Committee. Standard portable operating system interface for computer environments. IEEE Standard 1003.1-1988, 1988.

[17] ISO WG21. Programming languages — C. ISO/IEC Standard 9899:2011, Dec. 2011. A non-final but recent version is available at `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1539.pdf`, retrieved on 2016/8/24.

[18] J. Jelinek. RFC: ELF prelinker, 2001. Message to binutils mailing list, available at `http://www.sourceware.org/ml/binutils/2001-07/msg00057.html`.

[19] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2016.

[20] S. Kell. Towards a dynamic object model within Unix processes. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 224–239, 2015.

[21] A. Kennedy, N. Benton, J. B. Jensen, and P.-E. Dagand. Coq: The world's best macro assembler? In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 13–24, 2013.

[22] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 179–191, 2014.

[23] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[24] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[25] Linux Foundation. The Linux Standard Base 5.0. `http://refspecs.linuxfoundation.org/lsb.shtml` as re-

trieved on 2016/8/24.

[26] E. Machkasova and F. A. Turbak. A calculus for link-time compilation. In *Proceedings of the 9th European Symposium on Programming (ESOP)*, pages 260–274, 2000.

[27] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V Application Binary Interface, AMD64 architecture processor supplement, draft version 0.99.6. Available at `http://www.x86-64.org/documentation/abi.pdf`.

[28] J. S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Springer, 1996.

[29] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 175–188, 2014.

[30] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction (CC)*, pages 213–228, 2002.

[31] D. B. Orr, J. Bonn, J. Lepreau, and R. Mecklenburg. Fast and flexible shared libraries. In *Proceedings of the USENIX Summer Conference*, pages 237–251, 1993.

[32] E. Petrova. *Verifcation of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, 2007.

[33] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation (OSDI)*, page 24, 2000.

[34] A. Serra, N. Navarro, and T. Cortes. DITools: application-level support for dynamic extension and flexible composition. In *Proceedings of the USENIX Annual Technical Conference*, page 19, 2000.

[35] R. Shapiro, S. Bratus, and S. W. Smith. 'Weird machines' in ELF: A spotlight on the underappreciated metadata. In *Proceedings of the 7th USENIX Workshop on Offensive Technologies (WOOT)*, page 11, 2013.

[36] Y. Smaragdakis. Layered development with (Unix) dynamic libraries. In C. Gacek, editor, *Software Reuse: Methods, Techniques, and Tools*, volume 2319 of *Lecture Notes in Computer Science*, pages 33–45. Springer Berlin Heidelberg, 2002.

[37] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *Proceedings of the 47th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 275–287, 2015.

[38] The CompCert Development Team. CompCert manual. Available at `http://compcert.inria.fr/man/` as retrieved on 2016/8/24.

[39] The Santa Cruz Operation (SCO). System V Application Binary Interface, 10th June 2013. `http://www.sco.com/developers/gabi/latest/contents.html`.

[40] J. Ševčík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: a verified compiler for relaxed memory concurrency. *Journal of the ACM*, 60(22), 2013.

[41] P. Wang, S. Cuellar, and A. Chlipala. Compiler verification meets cross-language linking via data abstraction. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 675–690, 2014.

[42] J. B. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Proceedings of the 9th European Symposium on Programming (ESOP)*, pages 412–428, 2000.