Fold, a Dynamic Linker framework written in Rust

Master research project

by

Ludovic Mermod
Noé Terrier

Abstract

Fold is a framework to create Rust-based (dynamic) linkers, offering simple tools to design and implement new linkers. It provides a default modularized System V ABI linker, on top of which one can add incremental augments for custom purposes.

Instructor:	Prof. Édouard Bugnion
Teaching Assistants:	Charly Castes
Laboratory:	Data Center Systems Lab at EPFL
Date:	Spring semester 2025
Faculty:	School of Computer and Communication Sciences, EPFL

Contents

1 Motivation

When looking at the System landscape, it is clear that research is far ahead of actual implementations, as incorporating new technologies require to either merge them into the Linux Kernel or write a whole new OS. Both are very time-consuming and while the latter is more likely to succeed, it would most probably never get actually done due to the time it would take and the constraints for adding new features to Linux.

> "Systems Software Research is Irrelevant"
>
> — Rob Pike[1]

An interesting observation we can make on the design of Linux is that all processes, up to init itself, are launched by the system's dynamic loader[2]. This could be taken advantage of as changing the dynamic loader would allow executing user-defined code at the start of all processes. Furthermore, an ELF binary can specify the path of its loader, allowing it to pick an appropriate loader.

However, existing loaders like GNU's or Musl's are very complex pieces of code, intertwined with their respective standard library, making them hard to tweak. For example, when launching a process with GNU's loader, it first starts by linking itself with `libc`, and vice-versa as they both depend on each other, before finally linking the actual executable.

Based on these observations, we present Fold, a framework to easily create new dynamic linkers. It provides a basic linker implementation for usual executables and an API to add customized operations, similarly to LLVM's compiler framework[3].

2 Background

2.1 ELF

Before diving into Fold's inner working, let's first take a quick look at what an executable file looks like. ELF – *Executable and Linkable Format* – [4] is the format used for all executable files in a Linux environments. It is divided into four main parts: ELF header, program header table (PHT), content and section header table (SHT). The content itself is composed of segments and section, each having some extra metadata in, respectively, the program header table and the section header table. As depicted in Figure 1, it is important to note that segments and sections are two different "views" of the same content; a segment can overlap with a section and vice versa, but a segment cannot overlap with other segments. Segment carry information on the mapping of the ELF content in the virtual space and its protection, where section carry information on what the content is and how to interpret it (code, string table, `plt`...).

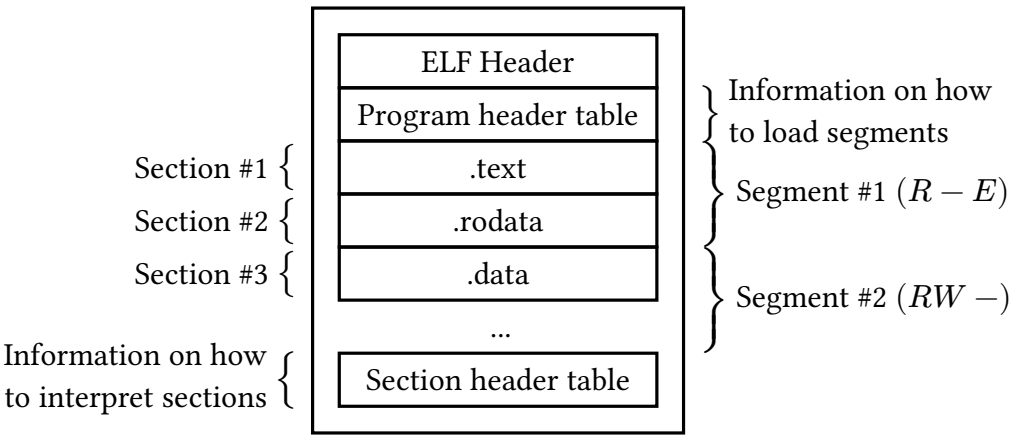

Figure 1: ELF File Structure

2.1.1 ELF header

The ELF header contains a few entries characterizing the file, starting with 16 magic bytes to identify the file as an ELF, position of the headers in the file, their size, the ABI used, etc. Interestingly, the version number has stayed at 1 for 50 years, although many variations have been designed and used (while staying compatible with older linkers).

2.1.2 Segments

The segments of the file contain data about how it should be executed, like the code, text and data segments, as well as the path of the interpreter (dynamic linker) to use. The most interesting segments in that project are those of the `LOAD` type, meaning that they need to be copied in to the process' address space. Their entry in the PHT also indicates the protection flags that need to be put on that segment (`R` for `.text`, `RE` for code, etc.) and the size of the segment in memory, which may differ from the size in the file if the segment ends with a sequence of zeros — in which case the dynamic linker needs to initialize the extra memory.

2.1.3 Sections

The sections describe how the file should be linked. Each entry in the table contains, among other things, a type, a name (or rather an index into a string table, see below), and a linked section.

The most important sections are the relocation sections (`.rela.*`), symbol tables and string tables:

- String tables (ST) contain all the string used in the file, for example for symbol names. The strings are stored as null-terminated sequences in the file. When a section references a string, it will actually hold the position of the string relative to the start of the ST section containing the string. The ST section itself can be easily identified as it is the one linked in the SHT.
- Symbol tables store the location of all symbols of the file. Symbols are used to identify functions, variable, and so on, when linking with dynamic libraries.
- Relocations tell the linker how to rewrite the executable's code such that it can interact with dynamically loaded libraries. More detail on this in Section 4.5.

2.2 System V ABI

OSDev wiki gives the following definition for System V ABI:

> "The System V Application Binary Interface is a set of specifications that detail calling conventions, object file formats, executable file formats, dynamic linking semantics, and much more for systems that complies with the X/Open Common Application Environment Specification and the System V Interface Definition. It is today the standard ABI used by the major Unix operating systems such as Linux, the BSD systems, and many others. The Executable and Linkable Format (ELF) is part of the System V ABI."
>
> — OSDev Wiki[5]

The design discussed later in Section 3 provides, among other things, a functional implementation of a loader that follow the System V ABI.

2.3 Linker workflow

A dynamic linker is a program able to transform an ELF file into an actual process. On execution request for an ELF file, the kernel open the file and looks for the path of the interpreter – *the dynamic linker* – required by the ELF inside the `.interp` section. It creates the process for the future executable, loads the dynamic linker into it and jumps to its entrypoint. The dynamic linker will identify the ELF file to load from the `argv`, open it, then parse it to retrieves loading and linking information. Finally, it will proceed to all the operations required by System V ABI in order to prepare the executable and pass the control flow to entrypoint.

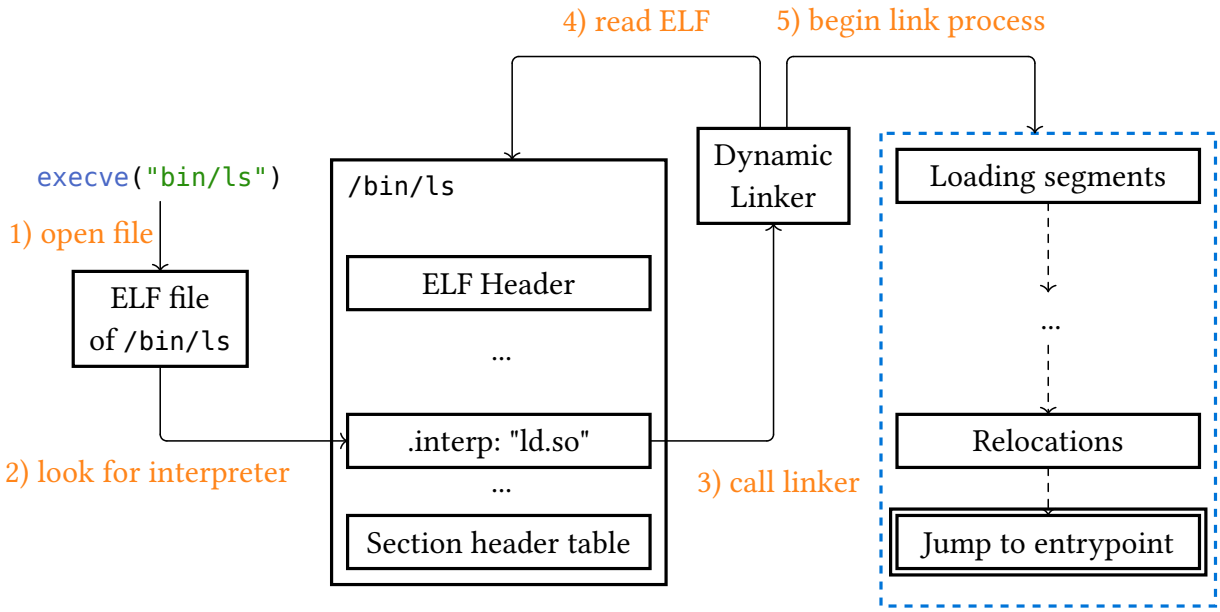

Figure 2: Flow of process creation

3 Fold Design

The idea behind Fold's design is similar to assembly lines: an object called the "manifold" is passed to several successive "modules", each of which modifies the manifold and/or the virtual space. Modules can communicate with each other though the manifold.

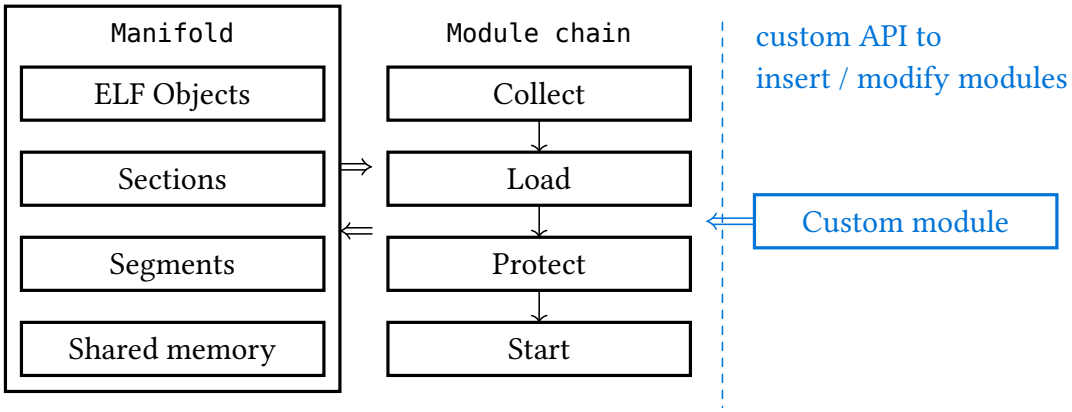

Figure 3: Fold structure

3.1 Manifold

The manifold structure shown in Code snippet 1 contains arrays of ELF objects, sections and segments, as well as a `ShareMap`. The latter is a structure able to store any datatype and is used to implement inter-module communication: a module can insert data into the map that can then be fetched and used by the following modules.

```
pub struct Manifold {
    pub objects: Arena<Object>,
    pub sections: Arena<Section>,
    pub segments: Arena<Segment>,
    pub shared: ShareMap,
    pub env: Env,
}
```

Code snippet 1: Manifold structure

For example, let's take a look at the first steps of the default System V module chain (Figure 4). First, the manifold is initialized with the ELF file of the binary to load. It then goes through the first module, which computes and loads the dependencies of the executable recursively, yielding a manifold with the initial ELF file plus all the dependencies ELFs, as well as an entry in the `ShareMap` containing the list of their paths. This is then passed over to the `Load` module which sets up the address space and load all the segments, from both the initial file and the dependencies at their respective addresses. It continues until the linker reaches the `Start` module which jumps to the exectuable's entry point and thus never return. Note that the linker does not know that the `Start` module is the end of the chain; this is only a consequence of the work of the module itself.

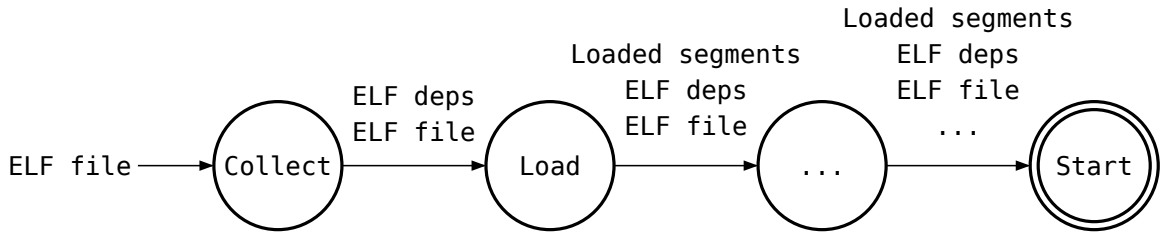

Figure 4: System V chain

3.2 Target selection

Modules can be applied to either the `Manifold`, objects, sections or segments. When registering a module in the chain, a filter can be added to specify which type of element it should match, as well as some more fine-grained selection to choose which specific elements to apply to. For example, in the chain above, `Start` would be applied the whole `Manifold` while `Load` would be invoked on all segments with the `PT_LOAD` tag.

When creating the chain of modules, filters are dissociated from the modules that they are applied to, allowing to compose modules more freely. For example if one wanted to modify how relocations are processed for the initial executable file but not its dependencies, they could register the usual relocation module with a filter excluding the executable object and a custom module only invoked on it.

4 System V Chain

We will now go through the implementation of the modules interacting with System V ABI[6] shown in Figure 4. These modules allow the default chain to link and execute various samples, from statically linked "Hello world!" up to a reduced yet fully functional build of SQLite.

As mentioned above, GNU's standard library is deeply intertwined with their linker, thus we moved away from this implementation and instead used Musl[7]'s standard library. It is much more simple and lightweight, thus making it way easier to interact with. We also slightly modified it such that it accepts being loaded by Fold instead of its own linker and added a few interface functions for compatibility with executables compiled with `gcc`.

We would also like to thank fasterthanlime and their incredible blog post "Making our own executable packer"[8], without which we could not have achieved such results.

4.1 Chain overview

Figure 5 shows the full default chain of System V modules, which is the one used in Fold's default build. As a first observation on the choices for Fold's design, the split into modules yields six modules with clearly defined tasks which need to communicate few data one to another through the manifold's shared memory. Some modules such as the collector (Section 4.2) can also leverage filters and shared memory to simplify their workflow by letting Fold call them multiple times and passing data from one invocation to another.

Precise filters can be assigned to most of the modules, simplifying the work done by the module itself. It is important to note that the filter for thread-local storage may be improved when implementing the complete behavior for this module (see Section 7).

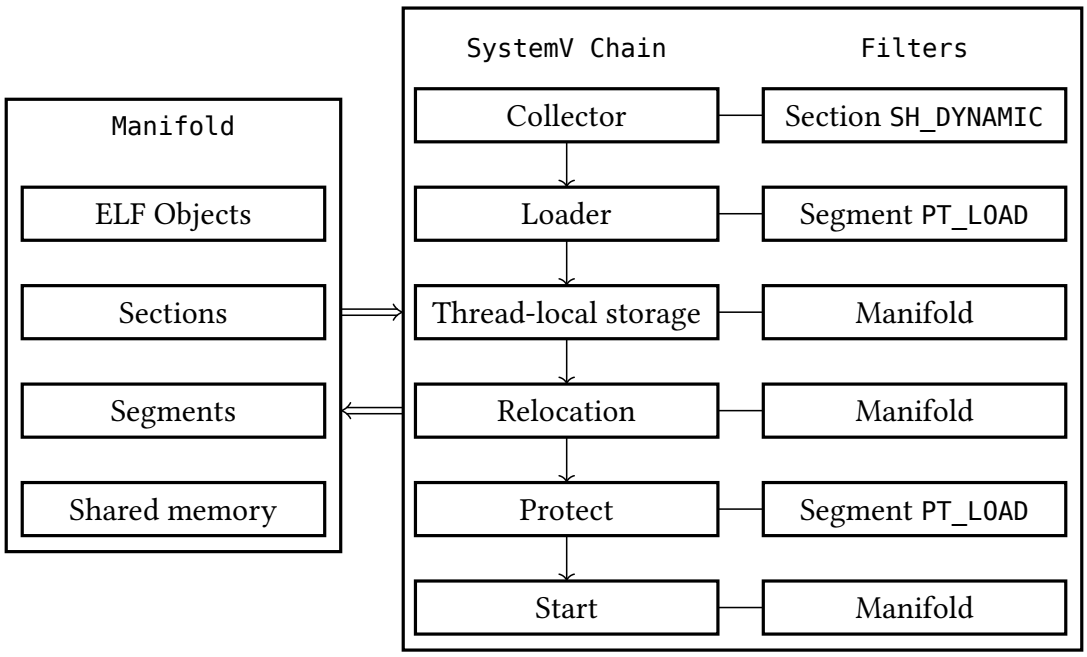

Figure 5: Default System V module chain

4.2 Collector

The first step in order to start the execution is to compute and load in memory all the dependencies of the ELF file. This can be achieved by iterating over the sections with the tag `SHT_DYNAMIC`, which contains record entries with a tag and a value. The tag `DT_NEEDED` indicates that the value it is attached to is a path to one of the ELF objects this file depends on; thus, filtering all the entries with that tag will yield all the file's dependencies.

Since the dependencies of the target ELF file can have their own dependencies, they need to be computed recursively while taking care to de-duplicate them. This can be easily achieved with the structure of the module: the collector is invoked once on the target ELF object and adds the direct dependencies to the manifold. Then, it will be invoked again on all the newly added ELF objects, resulting in a breadth-first iteration over the dependencies. To avoid duplicates, the module also stores in the manifold's shared structure a list of all the files it has already loaded, and subsequent invocations check that the dependencies they identified are not present in that list, then update it.

The last matter to take care of is the one of the differences between GNU's standard library and Musl's one. While GNU uses several object files for the standard C library (`libc.so.6`), the math library (`libm.so`) and so on, Musl puts all of them into a single `libc.so` file. This is handled by having the collector silently remaps all of `libc.so.6` to `libc.so` and drops all the dependencies that are bundled in the latter.

As said in Section 4.1, the iteration of sections can be simplified by using Fold. The framework will call the module on each element that matches the filter even if they were added after the start of the iteration. This means that the module can be registered with a filter for `SHT_DYNAMIC` sections, load the respective ELF files and update a set of the dependencies loaded in order to avoid duplicates.

4.3 Loader

Now that all the ELF objects are known, the linker needs to move on to the second step: setting up and populating the address space. As explained in Section 2.1.2, each object comes with a set of segment to be placed in the address space, indicated with the `PT_LOAD` tag in the program header table.

Each of these segments' entries features four important values: their physical address and size, and virtual equivalent. The physical address indicates where the segment is stored in the file (i.e. address relative to where the file is in memory), while the virtual address dictates where the segment needs to be stored for successful execution.

We must now distinguish two cases, depending on whether the object is dynamically linked or not. If it is, then the first segment to be loaded will ask for the address `0x0`, and the loader will substitute it for a random address (in our case, simply the address returned by `mmap`), and add this base address as a base offset to all the other segments loaded for this object. This is not needed for statically linked objects, as the virtual addresses of their segments is the exact location where they need to be placed — thus allocated with the `MAP_FIXED` flag.

One issue that may arise is that `mmap()` requires addresses and sizes aligned with the page size, which may not be the case for the addresses and sizes of the segments. To circumvent this, the module first computes the total size that the object will use in memory, i.e. the maximum value of virtual address plus virtual size, and call `mmap()` only once.

The physical size indicates the size of the segment in the file and not the size it will have in memory; the virtual size may be larger if the segments ends in zeros. In that case, after copying the segment, the module will initialize the differences with zeros.

Note that for now, the loaded segments are all mapped with read & write permission bits, necessary for the next modules. They will be updated with the actual permissions bits from the program header table later on, when all the modifications on the code will have been applied (Section 4.6).

4.4 Thread local storage

The thread-local storage is a part of memory referenced by the `fs` register and containing data related to the current thread, such as the thread control block (TCB) and some other data defined in the ELF. The specification of thread-local storage for ELF files[9] gives the following schema for the memory layout of TLS:

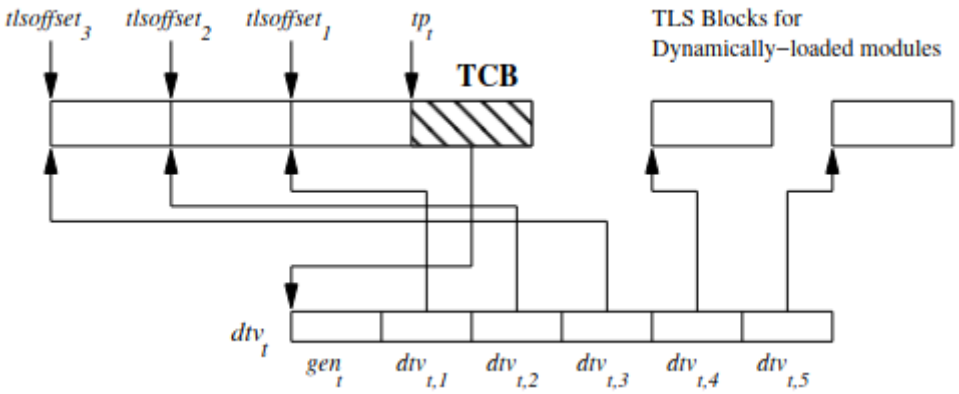

Figure 6: Thread-local storage memory layout (TLS specification[9], page 6)

The block including the TCB can simply be mmapped after computing the list of offsets to store and then must be initialized with the said offsets and the TCB struct[10]. As of now, the handling of the dynamic thread vector (`dtv`) and dynamic modules are not implemented in Fold as they are not required for the sample programs that were targeted, but a more complete implementation would require this to be completed (Section 7).

4.5 Relocation

A relocation is a modification of the content of loaded segments planned during compilation and resolved at link time, as it requires additional information like data from dependencies or even results of code execution. There is a large variety of relocation types, each with its own computation. They are mainly used to update `calls` to external library functions such that they hold the correct address of the function to jump to.

The relocation module processes sections with the tag `SHT_RELA`, each section containing an array of relocations. The order in which these entries must be handled is quite specific; it must be done in the order in which they appear in the object files but the objects must be processed in reverse order, meaning that objects loaded last (i.e. with no dependencies) are relocated first. The relocation entries hold an offset, a type and an extra value (also called addend). Depending on the type of the relocation, the operation to execute is different.

Here are some examples for x86 systems[6]:

Name	Value	Calculation
`R_X86_64_64`	1	S + A
`R_X86_64_JUMP_SLOT`	7	S
`R_X86_64_RELATIVE`	8	B + A
`R_X86_64_IRELATIVE`	37	indirect (B + A)

- **S** is the value of the symbol found inside the symbol table. It can be found in the linked symbol section and the index of the symbol is stored in the 32 MSBs of the addend.
- **A** is the addend from the symbol entry.
- **B** the base address of the object computed in Section 4.3.
- `indirect(X)` means that the resulting value is obtained by calling the code pointed by `X`.

Resolving the address of a symbol is a bit cumbersome. Symbols are accompanied by a `bind` value, which is either `LOCAL`, `GLOBAL` or `WEAK`. When searching for a given symbol, the linker must first look for `LOCAL` symbol present in the object for which the symbol is resolved, then `GLOBAL` symbols accross all objects and finally `WEAK` ones.

4.5.1 Jump slot relocation

While the `JUMP_SLOT` relocation may seem simple, its actual behavior is actually quite complex as it involves the procedure linkage and global offset tables (respectively PLT and GOT). In a nutshell, the relocation should not be processed with symbol resolution during the linking phase, but the addend should be used to relocate to the corresponding PLT entry. During the execution, when an external function is called for the first time, the program would give execution control back to the loader, which would resolve the symbol and update the GOT and then jump to the function, such that subsequent call would only have to read the GOT to jump to the correction location, without involving the loader. A more complete walk-through of the procedure can be found in section 5.2 of the ABI specification[6].

However, this behavior creates security issues, as the GOT, storing the addresses where to jump for external functions is writable. Modern linkers resolve the symbol at link-time and write it in the GOT, then marking it as read-only. In our implementation, we implemented an even simpler behavior, resolving all the symbols directly at the call site rather than in the GOT, completely bypassing the PLT and GOT. This choice was due mainly to time constraints, and needs to be addressed in future versions of the project (Section 7).

4.6 Protect

Once all the modifications on the segments' content are done, the linker must update the permission bits of the different segments to match those specified by each entry in the program header table. This is achieved using a simple `mprotect()` call, redefining the permissions of the segments one by one. Note that those permissions can only be set once the relocations are completed as they usually require modifying code, which is protected with read & execute bits.

4.7 Start

Final module of the chain. Before jumping into the main program, the stack still needs to be set. The module constructs the stack of the executable by pushing `args`, `env` and `auxv` into memory and correctly setting `rsp`. `args`, `env` and `auxv` are set by Linux and could be retrieved from the stack at the very beginning of fold execution. Finally, it jumps to the entry point of the ELF.

5 Case study

An interesting application of the modularized loader is that we can obviously extend the default chain of operation to add utilities. Here follows some examples of such a modified loader, and a demonstration of how simple it is to implement it.

5.1 Syscall filtering

From a security perspective, it could be interesting to reduce the number of syscalls a process have access to. The `seccomp` syscall exactly do that! It uses a filter implemented as an `eBPF`

program to restrict usage of syscalls. What we can do with Fold is to call `seccomp` before jumping to the entry point of our program.

To implement this, we can simply create a new Fold module, which performs a global operation on all the manifold. It constructs the filter with predefined syscalls and installs it inside the process with `seccomp`.

```
impl Module for Seccomp {
    fn name(&self) -> &'static str {
        "seccomp"
    }

    fn process_manifold(
        &mut self,
        _manifold: &mut fold::Manifold,
    ) -> Result<(), alloc::boxed::Box<dyn core::fmt::Debug>> {
        // Combine filters for write and exit
        let mut filters = build_seccomp_filter(&[SYS_WRITE, SYS_EXIT]);

        let mut prog = SockFprog {
            len: filters.len() as u16,
            filter: filters.as_mut_ptr(),
        };
        unsafe {
            // Requiered by SECCOMP_SET_MODE_FILTER
            syscall!(Sysno::prctl, PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)
                .map(|_| ())
                .map_err(|_| Box::from(SeccompError))?;

            // Install the filter using seccomp syscall
            syscall!(
                Sysno::seccomp,
                SECCOMP_SET_MODE_FILTER,
                0,
                &mut prog as *mut _ as usize
            )
            .map(|_| ())
            .map_err(|_| Box::from(SeccompError))
        }
    }
}
```

Code snippet 2: Module implementation of `Seccomp`

Once the module is created, we can define the entry point of our new loader, and augment the basic System V chain with our module. It means inserting the new module before the start module.

```
fn seccomp_chain(fold: Fold) -> Fold {
    fold.select("start")
        .before()
        .register("syscall restriction", Seccomp, Filter::manifold())
}
```

Code snippet 3: Main entry for seccomp-linker

Note that it is very easy to implement this directly in the linker, since it runs in the same process as the executable. Thus, calling `seccomp` from the linker's code will limit the resulting executable, without requiring to modify another process.

5.2 Inter-module communication

We can push the previous syscall filter idea further. For example, we could scan the object to detect the syscalls used and then restrict the process to only this set. The linker is a great place to do such analysis as it can observe the whole executable code and used symbols.

In order to do the scan, and to illustrate communication between modules, we chooe to create another module, at the beginning of the chain, that first collect symbols from the ELF and produce a set of syscalls to communicate to the `seccomp` filter module. The latter will retrieve it and create its filter from this.

```rust
impl Module for SysCollect {
    fn name(&self) -> &'static str {
        "syscall collect"
    }

    fn process_object(
        &mut self,
        obj: Handle<Object>,
        manifold: &mut Manifold,
    ) -> Result<(), Box<dyn core::fmt::Debug>> {
        let obj = &manifold[obj];

        let mut filter = vec![];

        // Combine filters for write and exit
        for symbol in obj.symbols(manifold) {
            if let Ok((_sym, name)) = symbol
                && name.to_string_lossy().contains("puts")
            {
                filter.push(SYS_WRITEV);
                filter.push(SYS_WRITE);
                filter.push(SYS_IOCTL);
                filter.push(SYS_EXIT_GROUP);
            }
        }

        log::info!("Identified syscall(s) needed: {filter:?}");

        manifold.shared.insert(SECCOMP_SYSCALL_FILTER, filter);

        Ok(())
    }
}
```

Code snippet 4: Module implementation for `SysCollect`

In this basic example, we detect only if `puts` is used and if so, add probably used syscall to the set.

The module can store this new set into the manifold shared map, with its own key:

```rust
manifold.shared.insert(SECCOMP_SYSCALL_FILTER, filter);
```

The previous module, which call `seccomp`, can retrieve this list by accessing to the manifold shared map:

```rust
let syscall_filter = manifold
    .shared
    .get(SECCOMP_SYSCALL_FILTER)
    .unwrap_or(&empty);
```

5.3 Function hooks

The goal of this example is to allow the injection of hooks before some of the dynamically linked functions. To be considered successful, these hooks should be invisible both to the program itself and to the libraries.

For each hook it wants to install, the linker creates two function, the hook itself and a trampoline function which is used to intercept the control flow of the program, call the hook and then resume the call to the target function. To ease the creation of the trampoline function, it is wrapped in a procedural macro (see Code snippet 5)

```
fn #trampoline_ident() {
    unsafe {
        ::core::arch::asm!(
            // Save the resolved address of the symbol in the stack. The actual
            // value writtenmust be changed by the linker.
            "mov rax,{}",
            "mov [rsp],rax",
            // Stores all the registers potentially containing arguments on the
            // stack. All other temporary registers are not used across the call
            // by the trampoline and thus do not need to be saved.
            "push rcx",
            "push rdx",
            "push rsi",
            "push rdi",
            "push r8",
            "push r9",
            // Call the hook
            "call {}",
            // Pops the arguments back into the corresponding registers.
            "pop r9",
            "pop r8",
            "pop rdi",
            "pop rsi",
            "pop rdx",
            "pop rcx",
            // Recovers the actual symbol to jump to, and jump there will passing
            // the return address of the current function
            // frame to the callee.
            "pop rbx",
            "mov rax,[rsp]",
            "jmp rbx",
            const 0xdeadbeefi64,
            sym #ident
        );
    }
}
```

Code snippet 5: Trampoline generation code

Due to its nature, the function is written entirely in assembly and is composed of 5 steps:

1. Store on the stack the address of the hijacked function. Having the linker to easily identify where it should resolve the symbol of the target without having to store this in a relocation.

2. Save the arguments registers. Those are callee-saved registers, hence the trampoline needs to take care of restoring them after calling the hook.
3. Call the hook. It is interesting to note that since arguments registers where unmodified since the entering the trampoline, they still hold the arguments passed to the hijacked function and can thus be read by the hook.
4. Restore the arguments registers.
5. Get the address of the hijacked function from the stack (set in step 1) and jump there. This means that the hijacked function will have the stack frame of the trampoline, with its return address which is the one the program needs to go back to after running the target function.

The overall execution flow of this function is very similar to a function such as the one shown in Code snippet 6 with tail-call optimization.

```
fn trampoline_puts(str: *const i8) {
  hook(str);
  puts(str);
}
```

Code snippet 6: Simple trampoline function

The linker adds an extra module after the relocation one to rewrite the relocation of the target functions to actually jump to the hook rather than the external library, and then update the hook's first `mov` instruction to hold the address of the hijacked function.

This implementation of hook is rather simple, allowing a single function to be targeted by a hook and the hook must be hard coded in the linker itself. A more complete implementation could instead look for the hooks in a new dedicated ELF section, and each hook could have several "entrypoints", with multiple `mov rax {}` instructions each followed by a jump to the `push rax` one which would allow to have several functions rewritten to different entrypoints.

It is also interesting to note that implementing such hooks aligns well with the linker's actual purpose as, outside of the trampoline function, the new module replace the hijacked functions' relocations by relocations to the trampoline and rewriting the trampoline's first `mov` is equivalent to a relocation to the hijacked function.

6 State of the project

Currently, the project features the System V modules for handling basic x86 executables and a full API to manipulate that chain. However, the existing modules are limited as they do not handle all relocation types and support for thread-local storage is sparse.

The default System V modules provided by the framework cover successfully the following types of ELF files: statically linked executables, position independent executables, dynamically linked executables, compiled c program using Musl's `libc`, and even a build-modified version of `sqlite3` built without multithreading and libc-dependent libraries.

There are also several examples that use the framework to achieve various purposes (Section 5). They were implemented without ever requiring modification of the design of the framework, showing that the design choices give enough freedom to easily implement new linker starting from the basic System V chain.

7 Future work

Although the design and main parts of the implementation are complete, there are still some challenges to address before the System V modules can be considered fully working. The two major ones are to complete the handling of thread-local storage (Section 4.4) and lazy processing of jump slot relocations (Section 4.5).

Along with these major milestones, some other improvements could be added, such as:

- Improving stack creation to reuse the initial stack of the process instead of creating a new one before jumping to the entrypoint:
- Unloading the code of the linker and the objects before starting the program.
- Symbol hash table for fast symbol lookup
- Others relocations not implemented

8 References

[1] R. Pike, [Online]. Available: http://herpolhode.com/rob/utah2000.pdf

[2] A. G. Charly Castes, "Dynamic Linkers Are the Narrow Waist of Operating Systems," Oct. 2023. [Online]. Available: https://charlycst.github.io/papers/dyn-linkers.pdf

[3] C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," in *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep. 2004. [Online]. Available: https://llvm.org/pubs/2004-09-22-LCPCLLVMTutorial.html

[4] T. Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification_1995." [Online]. Available: https://refspecs.linuxfoundation.org/elf/elf.pdf

[5] "OSDev Wiki." [Online]. Available: https://wiki.osdev.org/System_V_ABI

[6] M. G. J. H. A. J. M. M. H.J. Lu Michael Matz, "System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models)." [Online]. Available: https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build

[7] "Musl libc." [Online]. Available: https://musl.libc.org/

[8] fasterthanlime, "Making our own executable packer." [Online]. Available: https://fasterthanli.me/series/making-our-own-executable-packer

[9] U. Drepper, "ELF Handling For Thread-Local Storage." [Online]. Available: https://www.akkadia.org/drepper/tls.pdf

[10] "Thread control block head structure." [Online]. Available: https://course.khoury.northeastern.edu/cs5600f15/dmtcp/structtcbhead__t.html