ELFlex: A Trampoline-Based Binary Instrumentation Tool - Design Document

Overview

This document outlines the design of a binary instrumentation tool that modifies ELF executables to inject custom monitoring and behavior hooks at runtime. The key architectural challenge is inserting new code into an existing, compiled binary without breaking its structure or execution flow, requiring precise manipulation of executable formats, instruction patching, and dynamic code generation.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

> **Milestone(s):** 1 (Instruction-level Patching), 2 (Function Hooking with Trampolines)

Context and Problem Statement

Binary instrumentation is the practice of modifying compiled executable programs to insert custom monitoring code, performance measurements, security checks, or behavior modifications. This capability serves critical needs across software engineering: security researchers analyze malware behavior, performance engineers profile hot paths without source code, reverse engineers add debugging facilities to closed-source software, and developers implement runtime monitoring in production systems. However, modifying compiled binaries presents formidable technical challenges that this design document aims to systematically address.

Mental Model: The Building Blueprint and the Renovation Crew

Imagine a compiled ELF binary as the **complete architectural blueprint and construction plans for a complex building**. This blueprint contains not only the floor plans (the program's logic) but also the electrical wiring diagrams (data flows), structural specifications (memory layout), and even the precise locations where construction crews must begin work (function entry points). Like a blueprint, the binary is precise and unforgiving — moving a wall requires recalculating every measurement that references it, and adding new wiring requires understanding exactly how it connects to the existing infrastructure without causing short circuits.

Our instrumentation tool, **ELFlex**, serves as the **specialized renovation crew** tasked with adding new surveillance cameras (monitoring hooks) and safety systems (security checks) to this already-constructed building. The crew faces three fundamental constraints:

1. **The building must remain functional during renovations** — We cannot demolish walls to rebuild them from scratch; we must work within the existing structure.
2. **The original blueprint must remain valid** — Our modifications must produce an updated but equally valid blueprint (ELF file) that future tools can still understand.
3. **We cannot ask the original architects** — We lack access to the source code or build instructions; we only have the finished blueprint and the constructed building itself.

This renovation crew analogy extends to specific instrumentation operations:

Renovation Task	Binary Instrumentation Equivalent	Core Challenge
Adding a new camera in a hallway	Inserting a hook at function entry	Redirecting traffic without blocking the hallway
Running new wiring through existing walls	Injecting new code sections	Finding space without interfering with structural elements
Temporarily diverting foot traffic	Runtime patching of a running process	Making changes while people are still using the building
Updating the blueprint after changes	Maintaining ELF structural integrity	Ensuring all cross-references in the updated document remain accurate

> **Key Insight:** Binary instrumentation is fundamentally a **constraint-satisfaction problem**. Every modification must satisfy the dual constraints of (1) preserving the original program's functionality and (2) maintaining the ELF format's structural validity, all while working with only the compiled artifact as input.

The Core Problem: Modifying a Running (or Static) Machine

At its core, binary instrumentation faces three intertwined technical challenges that distinguish it from source-level modification:

1. The Self-Modifying Code Paradox

Modern computer architectures and operating systems assume code is **immutable during execution**. The x86-64 instruction set includes features optimized for this assumption (instruction cache, read-only code pages), creating a paradox when we need to modify that same code. The system actively resists the very modifications we need to make.

Technical manifestation: Code pages are typically mapped with read-execute (`PROT_READ | PROT_EXEC`) but not write (`PROT_WRITE`) permissions. To modify code, we must:

1. Temporarily change page protections (via `mprotect()`)
2. Perform the modification
3. Restore original protections
4. Flush the instruction cache (on architectures where required)

For static binary rewriting (Milestone 1-3), this occurs at the file level before execution. For runtime patching (Milestone 4), we must perform this delicate operation on a live process without causing crashes or undefined behavior.

2. Position-Dependent vs. Position-Independent Code

Compiled binaries contain two categories of code references that behave differently when we move or modify code:

Reference Type	Description	Example	Challenge for Instrumentation
Absolute References	Direct memory addresses hardcoded in instructions	`mov rax, 0x400500` (load from fixed address)	Break if we shift code to different addresses
Relative References	Offsets calculated from the current instruction pointer	`call 0xffffff80` (jump back 128 bytes)	May need recalculation if we insert code between caller and target
RIP-Relative References	x86-64 specific: offsets from the instruction pointer	`mov rax, [rip+0x200]` (access data 512 bytes ahead)	Particularly tricky when relocating instruction sequences

The most pernicious challenge arises with **RIP-relative addressing**, a common x86-64 optimization where instructions reference data relative to the current instruction pointer. When we extract and relocate a function's prologue to a trampoline (Milestone 2), any RIP-relative instructions within that prologue will now calculate incorrect addresses unless we apply fixups.

> **Design Principle:** All injected code must be **position-independent** (PIC) or carefully relocated to account for its final load address. This principle applies both to trampolines (which we generate) and to injected hook functions (which we compile separately).

3. ELF Structural Integrity

The ELF (Executable and Linkable Format) specification defines a precise structure with interdependent offsets and sizes. Consider this simplified view of ELF components and their relationships:

Component	Purpose	Interdependencies
ELF Header	Identifies file as ELF and points to other structures	Contains offsets to section and program header tables
Program Headers	Define memory segments for the loader	Reference file offsets and sizes of segment contents
Section Headers	Describe sections (debugging, linking)	Optional for execution but useful for analysis
.text Section	Contains executable code	Must be within a `PT_LOAD` segment marked executable
Symbol Table	Maps names to addresses (optional)	Entries reference section indices and offsets within sections

The fundamental integrity challenge is that **changing one component often requires updating multiple dependent components**. For example:

- Adding a new code section requires:
  1. Adding a section header entry
  2. Possibly adding a program header for a new `PT_LOAD` segment
  3. Updating the ELF header's section count
  4. Shifting all subsequent content in the file
  5. Updating all file offsets in headers that reference moved content

This interdependency creates a fragile balancing act where a single miscalculation produces a corrupted, unloadable binary.

4. State Preservation During Redirection

When we intercept function calls via trampolines, we temporarily hijack the CPU's execution flow. During this hijacking, we must preserve the **complete architectural state** so the original function executes exactly as if it were called directly. This includes:

State Component	Preservation Requirement	Risk if Not Preserved
General-purpose registers	Save before hook, restore after	Hook modifies values the original function depends on
Flags register	Preserve across hook execution	Original function's conditional branches behave incorrectly
Stack pointer	Maintain proper alignment	Could cause segmentation faults or ABI violations
Red zone (x86-64 ABI)	Avoid overwriting 128 bytes below stack pointer	Corruption of local variables in leaf functions

The trampoline design must account for all these preservation requirements while also providing the hook function access to the information it needs (function arguments, return address).

Existing Approaches and Trade-offs

Several established approaches to binary instrumentation exist, each with different trade-offs. Understanding these alternatives helps clarify why we chose the static binary rewriting approach for ELFlex.

Option 1: Static Binary Rewriting (Our Approach)

How it works: Modifies the on-disk ELF file before execution, adding instrumentation code directly to the binary. This is what ELFlex implements across Milestones 1-3.

Aspect	Characteristics
Overhead	Zero runtime overhead after initial load (instrumentation is part of binary)
Compatibility	Works with any ELF loader; no special runtime requirements
Complexity	High implementation complexity (ELF manipulation, relocation fixups)
Use Cases	Permanent instrumentation, security hardening, performance profiling tools
Limitations	Cannot instrument dynamically loaded libraries after process start

ADR: Static vs. Dynamic Instrumentation

Context: We need to choose a foundational approach for our instrumentation tool that balances implementation complexity with practical utility for learners.

Options Considered:

1. **Pure static rewriting** (modify on-disk binaries before execution)
2. **Dynamic binary instrumentation (DBI)** (intercept execution at runtime via just-in-time compilation)
3. **Hybrid approach** (static rewriting with runtime code generation)

Decision: Pure static binary rewriting for Milestones 1-3, with runtime patching (via ptrace) as a separate capability in Milestone 4.

Rationale:

1. **Educational value:** Static rewriting requires deep understanding of ELF format, instruction encoding, and relocation — foundational knowledge for systems programming.
2. **Incremental complexity:** We can build from simple instruction patching (Milestone 1) to full function hooking (Milestone 2) to code injection (Milestone 3) in manageable steps.
3. **Clear boundaries:** Each milestone has well-defined inputs (ELF file) and outputs (modified ELF file), making testing and verification straightforward.
4. **Practical utility:** Many real-world tools (like security hardening tools) use static rewriting, making the skills directly applicable.

Consequences:

- Positive: Learners gain deep understanding of executable file formats and low-level code modification.
- Negative: Cannot instrument dynamically loaded libraries without runtime component (addressed in Milestone 4).
- Trade-off: More complex implementation than DBI for simple hooking, but more educational value.

Option 2: Dynamic Binary Instrumentation (DBI)

How it works: Uses a runtime engine to intercept execution of the target program, disassembling and instrumenting code as it runs. Examples include DynamoRIO, Intel Pin, and Valgrind.

Aspect	Characteristics
Overhead	High (2-10x slowdown) due to just-in-time compilation and analysis
Compatibility	Requires runtime engine; may not work with all system calls or threading models
Complexity	Extremely high (JIT compilation, code cache management, exception handling)
Use Cases	Research, dynamic analysis, fuzzing, memory checking
Advantages	Can instrument dynamically loaded code, handle self-modifying code

Comparison Table: Static Rewriting vs. DBI

Criteria	Static Rewriting (ELFlex)	Dynamic Binary Instrumentation
Implementation Complexity	High (ELF manipulation)	Extremely high (JIT, code cache)
Runtime Overhead	Zero after load	Significant (2-10x slowdown)
Startup Time	Normal process startup	Slower (engine initialization)
Code Coverage	Only statically linked code	All code (including dynamic libraries)
Tool Portability	Self-contained modified binary	Requires runtime engine installation
Educational Focus	File formats, assembly, patching	Compilation, optimization, runtime systems

Option 3: Kernel-Based Instrumentation

How it works: Uses operating system kernel facilities to monitor or intercept system calls and process events. Examples include Linux `ftrace`, `kprobes`, and eBPF.

Aspect	Characteristics
Overhead	Very low (in-kernel instrumentation)
Compatibility	Kernel version dependent, requires root privileges
Complexity	Moderate (kernel module development)
Use Cases	System-wide monitoring, performance analysis, security auditing
Limitations	Cannot easily instrument userspace function logic, only system boundaries

Why not this approach for ELFlex: Kernel-based instrumentation operates at a different abstraction level — it intercepts system calls and kernel events but cannot easily modify userspace function logic within a process. Our goal is to learn about binary modification and function hooking specifically, which requires working at the process memory level rather than the kernel interface.

Option 4: Library Interposition (LD_PRELOAD)

How it works: Uses the dynamic linker's `LD_PRELOAD` mechanism to replace library function calls with wrapper functions. This is a simpler, userspace-only approach.

Aspect	Characteristics
Overhead	Minimal (function call overhead)
Compatibility	Only works with dynamically linked functions
Complexity	Low (standard shared library development)
Use Cases	Wrapping library calls, debugging malloc/free
Limitations	Cannot instrument static functions or main executable code

Why not this approach for ELFlex: While `LD_PRELOAD` is excellent for intercepting library calls, it cannot hook functions within the main executable or statically linked code. More fundamentally, it doesn't teach the low-level binary manipulation skills that are our primary educational objective.

> **Key Insight:** Each instrumentation approach represents a different point in the design space balancing **implementation complexity**, **runtime overhead**, and **instrumentation granularity**. Our choice of static binary rewriting prioritizes educational value and foundational knowledge, accepting the complexity of ELF manipulation as a worthwhile learning challenge.

The Path Forward: From Instruction Patching to Runtime Instrumentation

Given these challenges and trade-offs, ELFlex adopts a **progressive implementation strategy** across four milestones:

1. **Instruction-level Patching** (Milestone 1): Learn the fundamentals of modifying executable code while maintaining ELF integrity.
2. **Function Hooking with Trampolines** (Milestone 2): Build on patching to redirect function execution flow while preserving program state.
3. **Code Injection** (Milestone 3): Extend the tool to add entirely new code sections, enabling complex instrumentation logic.
4. **Runtime Patching via ptrace** (Milestone 4): Apply these techniques to running processes, bridging static and dynamic instrumentation.

This progression ensures each new concept builds on previously mastered skills, with each milestone addressing specific aspects of the core problem described above.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option
ELF Parsing	Manual parsing using ELF specification	`libelf` or similar library
Instruction Encoding	Hardcoded opcodes for simple patches	Capstone/Zydis for disassembly, Keystone for assembly
Runtime Patching	Basic ptrace with manual memory operations	`libelfin` for DWARF debugging info
Build System	Makefile with explicit rules	CMake with dependency management

Recommended File/Module Structure

Given the project's educational focus and C as the primary language, we recommend this directory structure:

```
elflex/
├── include/                # Public header files
│   ├── elflex.h           # Main API header
│   ├── elf_parser.h       # ELF parsing structures and functions
│   ├── patch.h            # Patching structures and functions
│   ├── trampoline.h       # Trampoline generation
│   ├── inject.h           # Code injection
│   └── runtime.h          # Runtime patching via ptrace
├── src/                   # Implementation source files
│   ├── main.c             # Command-line interface
│   ├── elf_parser.c       # ELF loading and parsing (Milestone 1)
│   ├── patch.c            # Instruction patching (Milestone 1)
│   ├── trampoline.c       # Trampoline generation (Milestone 2)
│   ├── inject.c           # Code injection (Milestone 3)
│   ├── runtime.c          # Runtime patching (Milestone 4)
│   └── utils.c            # Utility functions (NOP generation, etc.)
├── tests/                 # Test binaries and test code
│   ├── test_programs/     # Simple test ELF files
│   ├── unit_tests.c       # Unit tests for components
│   └── integration_tests.sh # Integration test scripts
├── payloads/              # Example hook functions to inject
│   └── example_hook.c     # Example C code for injection
├── Makefile               # Build configuration
└── README.md              # Project documentation
```

Infrastructure Starter Code

Here's complete, working starter code for basic ELF header definitions and utility functions:

```c
/* include/elf_definitions.h */

#ifndef ELF_DEFINITIONS_H

#define ELF_DEFINITIONS_H


#include <stdint.h>


/* Basic ELF type definitions (64-bit little-endian focus) */

typedef uint64_t Elf64_Addr;

typedef uint64_t Elf64_Off;

typedef uint16_t Elf64_Half;

typedef uint32_t Elf64_Word;

typedef int32_t  Elf64_Sword;

typedef uint64_t Elf64_Xword;

typedef int64_t  Elf64_Sxword;


/* ELF identification */

#define EI_NIDENT 16


/* ELF file header */

typedef struct {

    unsigned char e_ident[EI_NIDENT];

    Elf64_Half    e_type;

    Elf64_Half    e_machine;

    Elf64_Word    e_version;

    Elf64_Addr    e_entry;

    Elf64_Off     e_phoff;

    Elf64_Off     e_shoff;

    Elf64_Word    e_flags;

    Elf64_Half    e_ehsize;

    Elf64_Half    e_phentsize;

    Elf64_Half    e_phnum;

    Elf64_Half    e_shentsize;

    Elf64_Half    e_shnum;

    Elf64_Half    e_shstrndx;

} Elf64_Ehdr;


/* Section header */

typedef struct {
```

```
    Elf64_Word    sh_name;
    Elf64_Word    sh_type;
    Elf64_Xword   sh_flags;
    Elf64_Addr    sh_addr;
    Elf64_Off     sh_offset;
    Elf64_Xword   sh_size;
    Elf64_Word    sh_link;
    Elf64_Word    sh_info;
    Elf64_Xword   sh_addralign;
    Elf64_Xword   sh_entsize;
} Elf64_Shdr;

/* Program header */
typedef struct {
    Elf64_Word    p_type;
    Elf64_Word    p_flags;
    Elf64_Off     p_offset;
    Elf64_Addr    p_vaddr;
    Elf64_Addr    p_paddr;
    Elf64_Xword   p_filesz;
    Elf64_Xword   p_memsz;
    Elf64_Xword   p_align;
} Elf64_Phdr;

/* Symbol table entry */
typedef struct {
    Elf64_Word    st_name;
    unsigned char st_info;
    unsigned char st_other;
    Elf64_Half    st_shndx;
    Elf64_Addr    st_value;
    Elf64_Xword   st_size;
} Elf64_Sym;

/* Section type values */
#define SHT_NULL      0
#define SHT_PROGBITS  1
```

```
#define SHT_SYMTAB    2
#define SHT_STRTAB    3
#define SHT_RELA      4
#define SHT_NOBITS    8
#define SHT_REL       9

/* Section flags */
#define SHF_WRITE     0x1
#define SHF_ALLOC     0x2
#define SHF_EXECINSTR 0x4

/* Program header type */
#define PT_LOAD       1

/* ELF magic number */
#define ELF_MAGIC "\x7f\x45\x4c\x46"

#endif /* ELF_DEFINITIONS_H */
```

```c
/* src/utils.c - Basic utility functions */

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <elf.h>

#include "elf_definitions.h"


/* Simple error handling utility */

void elf_error(const char *message) {

    fprintf(stderr, "ELF Error: %s\n", message);

    /* In a full implementation, we might want to track error codes */

}


/* Read file into buffer (caller must free) */

unsigned char* read_file(const char *filename, size_t *size) {

    FILE *file = fopen(filename, "rb");

    if (!file) {

        elf_error("Failed to open file");

        return NULL;

    }


    fseek(file, 0, SEEK_END);

    *size = ftell(file);

    fseek(file, 0, SEEK_SET);


    unsigned char *buffer = malloc(*size);

    if (!buffer) {

        elf_error("Memory allocation failed");

        fclose(file);

        return NULL;

    }


    if (fread(buffer, 1, *size, file) != *size) {

        elf_error("Failed to read entire file");

        free(buffer);

        fclose(file);
```

```
        return NULL;
    }

    fclose(file);
    return buffer;
}

/* Write buffer to file */
int write_file(const char *filename, unsigned char *buffer, size_t size) {
    FILE *file = fopen(filename, "wb");
    if (!file) {
        elf_error("Failed to create output file");
        return 0;
    }

    if (fwrite(buffer, 1, size, file) != size) {
        elf_error("Failed to write entire file");
        fclose(file);
        return 0;
    }

    fclose(file);
    return 1;
}

/* Check if buffer contains a valid ELF file */
int is_valid_elf(const unsigned char *buffer, size_t size) {
    if (size < sizeof(Elf64_Ehdr)) {
        return 0;
    }

    const Elf64_Ehdr *ehdr = (const Elf64_Ehdr*)buffer;

    /* Check magic number */
    if (memcmp(ehdr->e_ident, ELF_MAGIC, 4) != 0) {
        return 0;
```

```
    }

    /* Check class (64-bit) */
    if (ehdr->e_ident[EI_CLASS] != ELFCLASS64) {
        elf_error("Only 64-bit ELF files are supported");
        return 0;
    }

    /* Check data encoding (little-endian) */
    if (ehdr->e_ident[EI_DATA] != ELFDATA2LSB) {
        elf_error("Only little-endian ELF files are supported");
        return 0;
    }

    return 1;
}
```

Core Logic Skeleton Code

For the main ELF loading component (foundational for Milestone 1):

```c
/* src/elf_parser.c - ELF loading and parsing skeleton */

#include "elf_parser.h"

#include "elf_definitions.h"

#include <stdlib.h>

#include <string.h>


/*

 * Load an ELF file into an in-memory representation for modification

 *

 * Parameters:

 *   filename: Path to the ELF file to load

 *

 * Returns:

 *   Pointer to elf_binary_t structure on success, NULL on failure

 *

 * Memory Management:

 *   Caller must call elf_binary_free() on the returned pointer

 */

elf_binary_t* elf_load(const char *filename) {

    /* TODO 1: Read the entire file into a buffer using read_file() utility */

    /* TODO 2: Validate it's a valid ELF file using is_valid_elf() */

    /* TODO 3: Allocate and initialize elf_binary_t structure */

    /* TODO 4: Parse ELF header and store basic information */

    /* TODO 5: Load section headers (if present) into sections array */

    /* TODO 6: Load program headers into segments array */

    /* TODO 7: Find and mark the .text section (executable code section) */

    /* TODO 8: Load symbol table if present for function names */

    /* Hint: Keep the original buffer - we'll modify it in place later */



    return NULL; /* Replace with actual implementation */

}


/*

 * Find a section by name in the loaded ELF binary

 *

 * Parameters:
```

```
 *   binary: Pointer to loaded ELF binary
 *   name: Section name to find (e.g., ".text", ".symtab")
 *
 * Returns:
 *   Pointer to section_t structure if found, NULL otherwise
 */
section_t* elf_find_section(elf_binary_t *binary, const char *name) {
    /* TODO 1: Iterate through binary->sections array */
    /* TODO 2: For each section, get its name from section header string table */
    /* TODO 3: Compare with target name, return section if match found */
    /* TODO 4: Handle case where string table isn't loaded yet */

    return NULL; /* Replace with actual implementation */
}

/*
 * Get the address of a symbol by name
 *
 * Parameters:
 *   binary: Pointer to loaded ELF binary
 *   symbol_name: Name of the symbol to find
 *
 * Returns:
 *   Virtual address of the symbol if found, 0 otherwise
 */
uint64_t elf_get_symbol_address(elf_binary_t *binary, const char *symbol_name) {
    /* TODO 1: Locate the symbol table section (.symtab) */
    /* TODO 2: Locate the associated string table (.strtab) */
    /* TODO 3: Iterate through symbol table entries */
    /* TODO 4: For each symbol, get name from string table, compare with target */
    /* TODO 5: Return st_value (virtual address) if match found */
    /* Note: st_value may need adjustment based on load address */

    return 0; /* Replace with actual implementation */
}
```

```c
/*
 * Free all resources associated with an ELF binary
 *
 * Parameters:
 *   binary: Pointer to binary to free (can be NULL)
 */
void elf_binary_free(elf_binary_t *binary) {
    /* TODO 1: Check if binary is NULL, return early if so */
    /* TODO 2: Free sections array if allocated */
    /* TODO 3: Free segments array if allocated */
    /* TODO 4: Free original file buffer if allocated */
    /* TODO 5: Free binary structure itself */
    /* Note: Be careful of double-free if structures share memory */
}
```

Language-Specific Hints for C

1. **Memory Management**: Use a consistent ownership model. The `elf_binary_t` should own all its substructures. Consider using flexible array members for arrays of headers.
2. **Error Handling**: Since we're not using exceptions in C, establish a clear error reporting strategy. Options include:
   - Return NULL/0 for errors with error messages printed to stderr
   - Use an error code return with an out parameter for results
   - Maintain a global error state (like `errno` but for our library)
3. **Portability Considerations**:
   - Use fixed-width types (`uint32_t` , `uint64_t`) instead of architecture-dependent types
   - Handle endianness explicitly (we're assuming little-endian for x86-64)
   - Use `#pragma pack` or compiler attributes to ensure struct layout matches ELF specification
4. **Performance Considerations**:
   - For large binaries, consider mmap instead of reading entire file into memory
   - Cache frequently accessed sections (like symbol table) in separate structures
   - Use hash tables for symbol name lookups if performance becomes critical

Milestone Checkpoint for Context Section

Even before implementing Milestone 1, you can validate your understanding of the context:

Validation Test:

1. Create a simple test program:

```c
/* test_programs/hello.c */

#include <stdio.h>



int main() {

    printf("Hello, World!\n");

    return 0;

}
```

2. Compile it: `gcc -o hello hello.c -no-pie` (disable PIE for simpler addresses)
3. Examine the ELF structure:

```
readelf -h hello          # View ELF header
readelf -S hello          # View sections
readelf -s hello          # View symbols
objdump -d hello          # Disassemble code
```

4. Manually locate:
   - The `.text` section offset and size
   - The `main` function symbol address
   - The instruction bytes at the start of `main`

Expected Understanding:

- You should be able to explain how the ELF header points to section headers
- You should understand the difference between file offsets and virtual addresses
- You should recognize common x86-64 instructions in the disassembly

Signs of Misunderstanding:

- Confusing virtual addresses with file offsets
- Not understanding why position-independent code complicates patching
- Not recognizing the challenges of maintaining ELF integrity after modifications

---

Goals and Non-Goals

> **Milestone(s):** 1 (Instruction-level Patching), 2 (Function Hooking with Trampolines), 3 (Code Injection), 4 (Runtime Patching via ptrace)

The ELFlex project exists to provide developers with a practical, hands-on introduction to binary instrumentation—a powerful but complex domain typically reserved for security researchers and advanced system developers. Before diving into architectural details, it's crucial to establish clear boundaries that define what the system will accomplish (goals) and what it deliberately excludes (non-goals). This clarity prevents scope creep, focuses development effort on achievable learning outcomes, and sets realistic expectations for the tool's capabilities.

Goals: What ELFlex Must Achieve

ELFlex aims to build a functional binary instrumentation tool that demonstrates four progressively complex techniques for modifying executable code. These goals map directly to the project's milestones and represent the minimum viable feature set that, when completed, will give developers working knowledge of how instrumentation tools work internally.

Primary Goal: Enable Practical Learning Through Implementation

The foremost objective isn't to build a production-grade instrumentation framework, but to create a **pedagogical tool** that makes binary instrumentation concepts tangible. Each milestone exposes a distinct layer of complexity:

1. **Instruction-Level Patching (Milestone 1)** - Learn to manipulate raw machine code within an ELF binary while maintaining structural integrity
2. **Function Hooking with Trampolines (Milestone 2)** - Understand how to intercept function calls by redirecting execution flow
3. **Code Injection (Milestone 3)** - Master adding entirely new code sections to compiled binaries
4. **Runtime Patching via ptrace (Milestone 4)** - Discover how to modify running processes without restarting them

> **Key Insight:** These goals form a dependency chain—each builds upon skills learned in previous milestones. You cannot implement runtime trampoline injection (Milestone 4) without understanding how trampolines work (Milestone 2), which requires basic patching abilities (Milestone 1).

Functional Goal 1: Static ELF Modification

ELFlex must provide comprehensive capabilities for analyzing and modifying ELF binaries on disk. This includes:

Capability	Description	Required for Milestone
ELF Parsing	Load and parse 64-bit ELF executables, extracting headers, sections, segments, and symbol tables	1, 2, 3
Text Section Manipulation	Locate the executable `.text` section, read its contents, and write modifications back	1, 2
Instruction Replacement	Replace machine instructions at specific offsets while handling size mismatches via NOP padding	1
Symbol Resolution	Locate functions by name via the symbol table (`dynsym` / `.symtab`) and calculate their virtual addresses	2
Section/Header Updates	Add new sections, update section and program headers, and maintain ELF structural validity	3

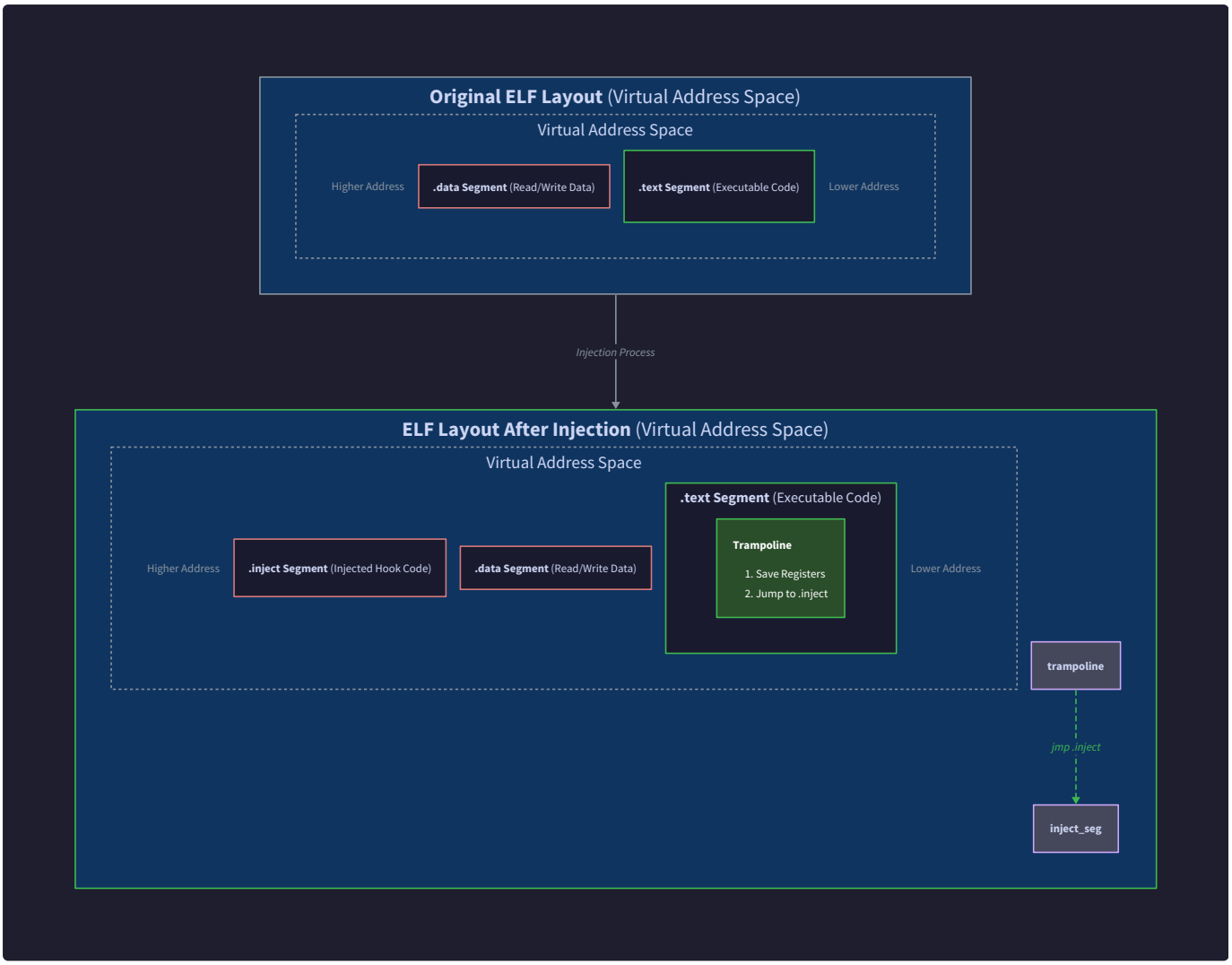
Functional Goal 2: Trampoline-Based Function Interception

The tool must implement the classic function hooking pattern using trampolines, which temporarily redirect execution to custom code:

Capability	Description	Key Challenge
Prologue Replacement	Overwrite the first few bytes of a target function with a jump to trampoline code	Must preserve enough bytes for a complete jump instruction (typically 5-14 bytes on x86-64)
Trampoline Generation	Generate position-independent code that saves registers, calls a hook function, restores registers, executes the relocated original prologue, then jumps to the remainder of the original function	Register preservation must be complete to avoid corrupting application state
RIP-Relative Fixups	Adjust instructions in the relocated prologue that use RIP-relative addressing (common on x86-64 for accessing data and function pointers)	Requires disassembling relocated instructions to identify those needing adjustment

Functional Goal 3: Code Injection and Symbol Resolution

ELFlex must inject new executable code into target binaries and resolve cross-references between injected and original code:



Capability	Description	Implementation Approach
New Section Creation	Add a new executable section (e.g., `.inject`) to the ELF file with proper alignment and permissions	Create new `Elf64_Shdr` and potentially a new `PT_LOAD` segment
Position-Independent Code	Inject code that can execute correctly regardless of its load address (or generate relocation entries if not PIC)	Use RIP-relative addressing exclusively or generate proper relocations
Cross-Reference Resolution	Allow injected code to call functions from the original binary by resolving symbol addresses at injection time	Parse symbol tables and patch call sites in the injected code
Build Pipeline Integration	Support compiling C functions separately and injecting their machine code into the target binary	External compilation with `gcc -fPIC -c` followed by ELF section extraction

Functional Goal 4: Runtime Process Instrumentation

The tool must instrument running processes using Linux's `ptrace` facility for dynamic analysis and debugging scenarios:

Capability	Description	Safety Consideration
Process Attachment	Attach to a running process using `PTRACE_ATTACH`, suspend its execution, and gain memory access	Must handle permission restrictions (e.g., root requirements)
Memory Manipulation	Read and write process memory using `PTRACE_PEEKDATA` / `PTRACE_POKEDATA`	Atomic operations on words, not arbitrary byte ranges
Code Page Protection	Change memory page permissions via `mprotect` (typically through syscall injection) to make code pages writable	Must restore original permissions after patching
Instruction Cache Coherence	Ensure modified code is visible to the CPU by flushing the instruction cache	Use `__builtin___clear_cache()` on affected memory ranges
Clean Detachment	Resume process execution and detach without leaving the process in a corrupted or stuck state	Handle pending signals and ensure trampolines are correctly installed

Quality Attributes (Non-Functional Goals)

Beyond functional capabilities, ELFlex should exhibit certain quality characteristics that make it effective as a learning tool:

Attribute	Description	Rationale
Clarity over Performance	Code should be readable and pedagogically sound, even if not optimally efficient	The primary audience is learners, not production users
Progressive Disclosure	Complexity should increase gradually across milestones, with each building on previous concepts	Prevents learners from being overwhelmed
Comprehensive Error Handling	Clear error messages that explain what went wrong and why, especially for common ELF manipulation mistakes	Helps learners debug their implementations
Cross-Platform Compatibility	Support for common x86-64 Linux distributions (Ubuntu, Fedora, Debian) with standard toolchains	Ensures broad accessibility for learners
Minimal External Dependencies	Prefer standard C libraries and POSIX APIs over specialized frameworks	Reduces setup complexity and dependency issues

> **Architecture Decision: Scope Definition Methodology Context:** Binary instrumentation is a vast field with many possible features and optimizations. As a pedagogical project, we needed clear criteria for what to include. **Options Considered:**
>
> 1. **Production-Feature Complete:** Implement all features a real instrumentation tool would need (relocations, exception handling, multi-architecture support)
> 2. **Concept-Minimal:** Implement only the absolute minimum needed to demonstrate each concept
> 3. **Learning-Optimized:** Select features that best illustrate core concepts while keeping implementation manageable **Decision:** Learning-Optimized scope (Option 3) **Rationale:** The project's primary purpose is education, not production use. Each feature was evaluated for its pedagogical value versus implementation complexity. Features that illustrate fundamental concepts (like RIP-relative fixups) were included, while optimizations (like jump table compression) were excluded. **Consequences:** The tool won't be suitable for production instrumentation tasks but will effectively teach the underlying principles. Learners can extend it later if needed.

Non-Goals: Explicit Exclusions

Equally important to defining what ELFlex *will* do is clarifying what it *won't* do. These exclusions prevent scope creep and focus development on the core learning objectives.

Non-Goal 1: Support for Non-ELF Binary Formats

ELFlex exclusively targets Linux ELF (Executable and Linkable Format) binaries. It will not support:

Format	Reason for Exclusion
Windows PE/COFF	Different header structure, relocation format, and ABI would double the codebase size
macOS Mach-O	Different segment/section model and dyld linking model
Linux a.out	Legacy format with limited relevance
Raw firmware/bootloaders	Lack of standard headers and symbol information

This limitation allows learners to deeply understand one format rather than superficially covering many. The concepts learned (trampolines, code injection) are transferable to other formats, but the implementation details differ.

Non-Goal 2: Full Dynamic Binary Instrumentation (DBI) Framework

ELFlex implements specific instrumentation techniques but is not a complete DBI framework like Intel Pin or DynamoRIO:

DBI Framework Feature	ELFlex Approach	Rationale
Just-in-Time Compilation	Pre-compiled injection only	JIT compilation adds enormous complexity beyond core instrumentation concepts
Comprehensive API	Minimal hook interface	Full APIs with callbacks, analysis routines, and instrumentation points are a separate domain
Transparent Execution	Explicit function hooking only	Tools like DynamoRIO intercept every basic block; ELFlex hooks specific functions
Optimization Passes	No code optimization	Performance optimization distracts from learning instrumentation mechanics

The distinction is between **binary rewriting** (statically modifying binaries) and **dynamic instrumentation** (JIT-based runtime modification). ELlex focuses on the former as a foundation.

Non-Goal 3: Kernel Module Instrumentation

ELFlex operates in user space only:

Kernel Instrumentation	User-Space Equivalent in ELFlex	Exclusion Reason
System call hooking	Function hooking in user-space libraries	Kernel modules require different safety mechanisms and have higher privilege requirements
Interrupt handling	Signal handling in user space	Hardware-level concepts are beyond the project's scope
Kernel object manipulation	Process memory manipulation via ptrace	Kernel APIs and data structures are OS-specific and complex

Kernel instrumentation introduces ring-0 privileges, different safety models, and hardware-specific considerations that would derail the focus on ELF manipulation.

Non-Goal 4: Obfuscation and Anti-Reversing Features

While binary instrumentation is often used in malware analysis and reverse engineering, ELFlex won't implement obfuscation or anti-debugging techniques:

Feature	Status	Reason
Code packing/encryption	Not implemented	Adds complexity without teaching instrumentation fundamentals
Anti-debugging tricks	Not implemented	Focus is on instrumentation, not evasion
Control flow flattening	Not implemented	Advanced transformation beyond basic hooking
Self-modifying code	Limited to trampolines	Full self-modification adds significant complexity

The tool is designed for analysis and learning, not for creating hardened binaries.

Non-Goal 5: Multi-Architecture Support

Initial implementation targets x86-64 only:

Architecture	Support Level	Reasoning
x86-64	Primary target	Most common learner environment, well-documented
ARM64 (AArch64)	Not supported initially	Different instruction set, calling convention, and relocation model
x86 (32-bit)	Not supported	Declining relevance, different ELF class (ELF32 vs ELF64)
RISC-V	Not supported	Emerging but not yet widespread in learner environments

Supporting multiple architectures would require abstracting instruction encoding, trampoline generation, and RIP-relative fixup logic. This abstraction layer would obscure the direct relationship between concepts and implementation that learners need.

> **Architecture Decision: Single-Architecture Focus Context:** Binary instrumentation requires architecture-specific knowledge (instruction sets, calling conventions, relocation types). Supporting multiple architectures would increase complexity significantly. **Options Considered:**
>
> 1. **x86-64 Only:** Implement all features for one architecture
> 2. **Architecture-Abstracted:** Create abstraction layers for instruction manipulation
> 3. **Multi-Architecture:** Implement parallel code paths for x86-64 and ARM64 **Decision:** x86-64 Only (Option 1) **Rationale:** 1) x86-64 is the most common architecture in learner environments. 2) Architecture abstraction would create a layer of indirection that obscures how instrumentation actually works. 3) The core concepts (trampolines, code injection) transfer to other architectures, even if implementation details differ. **Consequences:** Learners using ARM-based systems (like Apple Silicon Macs) will need x86-64 emulation or cross-compilation. However, they gain deeper understanding of one architecture rather than superficial knowledge of several.

Non-Goal 6: Comprehensive Relocation Support

ELFlex handles basic RIP-relative fixups for relocated prologue instructions but doesn't implement full relocation processing:

Relocation Type	ELFlex Support	Reason for Partial/No Support
RIP-relative in relocated code	Basic support in trampoline generator	Required for basic function hooking to work
Global offset table (GOT) updates	Not supported	Adds significant complexity for marginal learning value
Procedure linkage table (PLT)	Not supported	PLT/GOT resolution is a linking concept, not core instrumentation
Dynamic symbol resolution	Not supported	Runtime linking is orthogonal to static instrumentation

The tool assumes position-independent code for injected sections, avoiding the need for most relocations. When code must be relocated (like function prologues), only the most essential fixups are implemented.

Non-Goal 7: Graphical User Interface or High-Level Language Bindings

ELFlex is a command-line tool and library written in C:

Interface Type	ELFlex Approach	Rationale
Graphical UI	Command-line only	GUIs add frontend complexity unrelated to binary instrumentation
Python bindings	C API only	Foreign function interfaces are a separate topic
Web interface	Not provided	Network and web development are orthogonal skills
High-level instrumentation DSL	C configuration	Domain-specific languages for instrumentation exist (e.g., Pin's API) but obscure low-level details

The focus remains on understanding what happens at the machine code level, not on building user-friendly interfaces.

Non-Goal 8: Production-Grade Robustness and Security

While ELFlex should handle errors gracefully, it doesn't aim for the robustness level required of production tools:

Production Consideration	ELFlex Approach	Reasoning
Malformed input handling	Basic validation with clear errors	Production tools need extensive validation; learners benefit from seeing failures
Security hardening	Minimal (avoid obvious vulnerabilities)	Security is a separate concern from instrumentation mechanics
Performance optimization	Clarity over speed	Optimized code is often harder to understand
Comprehensive testing	Milestone-focused tests	Full test coverage would exceed project scope

The tool is designed for learning and experimentation, often with known binaries in controlled environments.

Non-Goal 9: Support for Shared Libraries (.so files) Modification

ELFlex primarily targets executables, not shared libraries:

Binary Type	Support Level	Reason
Static executables	Primary target	Self-contained, easier to manipulate
Dynamic executables	Supported	Require handling of dynamic segments
Shared libraries (.so)	Limited/experimental	Position-independent requirements, symbol visibility, and `DT_NEEDED` entries add complexity
Position-independent executables (PIE)	Supported (with limitations)	Common on modern systems, similar to shared libraries

While some techniques work on shared libraries, the additional complexity of `DT_NEEDED`, `DT_SYMTAB`, and strict PIC requirements would distract from core concepts. Learners can extend to shared libraries after mastering executables.

> **Key Insight:** These non-goals aren't limitations but deliberate design constraints that focus the project on its educational mission. By saying "no" to certain features, we can say "yes" to deeper understanding of the fundamentals. Each excluded area represents a potential future extension for learners who want to explore further after completing the core milestones.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option (for extension)
ELF Parsing	Manual header parsing using standard structs	libelf (full-featured library)
Instruction Disassembly	Manual decoding for specific instructions (jumps, calls)	Capstone Engine (full disassembler)
Code Generation	Hand-written assembly with inline asm or .s files	LLVM JIT or dynasm for runtime code generation
Process Manipulation	Direct ptrace syscalls	libptrace or higher-level debugging libraries
Build System	Simple Makefile with explicit dependencies	CMake or Meson for cross-platform builds

B. Recommended File/Module Structure

For the Goals/Non-Goals section, we're establishing the project's scope. The implementation will be organized as:

```
elflex/                         # Project root
├── src/                        # Core implementation
│   ├── elf/                    # ELF parsing and manipulation (Milestone 1, 2, 3)
│   │   ├── elf_loader.c        # `elf_load`, `elf_find_section`, etc.
│   │   ├── elf_loader.h        # Public interface for ELF operations
│   │   ├── elf_structs.h       # `elf_binary_t`, `section_t`, `segment_t`
│   │   └── elf_utils.c         # Helper functions for ELF manipulation
│   ├── patch/                  # Patching infrastructure (Milestone 1)
│   │   ├── patcher.c           # `plan_patch`, `apply_patch`
│   │   ├── patcher.h           # Patch planning interface
│   │   └── nop_generator.c     # `generate_nop_sled`
│   ├── trampoline/             # Trampoline generation (Milestone 2)
│   │   ├── trampoline.c        # `generate_trampoline`, `relocate_prologue`
│   │   ├── trampoline.h        # Trampoline interface
│   │   └── fixup.c             # `fixup_rip_relative` and other fixups
│   ├── inject/                 # Code injection (Milestone 3)
│   │   ├── injector.c          # `inject_section`, `allocate_code_cave`
│   │   ├── injector.h          # Injection interface
│   │   └── resolver.c          # `resolve_symbol_in_target`
│   ├── runtime/                # Runtime patching (Milestone 4)
│   │   ├── ptrace_patcher.c    # `attach_to_process`, `write_process_memory`
│   │   ├── ptrace_patcher.h    # Runtime patching interface
│   │   └── memory_protect.c    # `make_page_writable`
│   └── common/                 # Shared utilities
│       ├── utils.c             # `read_file`, `write_file`, `is_valid_elf`
│       ├── utils.h             # Utility function declarations
│       └── errors.h            # Error codes and `elf_error` function
├── include/                    # Public headers (if building as library)
│   └── elflex.h                # Main public API header
├── tools/                      # Command-line tools
│   ├── elflex-cli.c            # Main CLI tool implementing all milestones
│   └── test-inject.c           # Example code to inject into binaries
├── test-binaries/              # Simple test programs for instrumentation
│   ├── hello.c                 # Basic "Hello World" for testing
│   ├── functions.c             # Multiple functions for hooking tests
│   └── Makefile                # Builds test binaries
├── tests/                      # Unit and integration tests
├── examples/                   # Example usage scripts and configurations
├── Makefile                    # Main build file
└── README.md                   # Project documentation
```

C. Infrastructure Starter Code

Here's a complete, working utility module that handles file I/O and basic ELF validation—prerequisites for all milestones that don't distract from the core learning goals:

```c
/* src/common/utils.c - Complete implementation */

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <sys/stat.h>

#include "utils.h"

#include "errors.h"


/* Read entire file into buffer */

unsigned char* read_file(const char* filename, size_t* size) {

    if (!filename || !size) {

        elf_error("read_file: invalid parameters");

        return NULL;

    }



    int fd = open(filename, O_RDONLY);

    if (fd < 0) {

        elf_error("read_file: cannot open file");

        return NULL;

    }



    struct stat st;

    if (fstat(fd, &st) < 0) {

        close(fd);

        elf_error("read_file: cannot get file size");

        return NULL;

    }



    *size = st.st_size;

    unsigned char* buffer = malloc(*size);

    if (!buffer) {

        close(fd);

        elf_error("read_file: memory allocation failed");
```

```
        return NULL;
    }

    ssize_t bytes_read = read(fd, buffer, *size);
    close(fd);

    if (bytes_read != (ssize_t)*size) {
        free(buffer);
        elf_error("read_file: incomplete read");
        return NULL;
    }

    return buffer;
}

/* Write buffer to file */
int write_file(const char* filename, unsigned char* buffer, size_t size) {
    if (!filename || !buffer) {
        elf_error("write_file: invalid parameters");
        return -1;
    }

    int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0755);
    if (fd < 0) {
        elf_error("write_file: cannot create file");
        return -1;
    }

    ssize_t bytes_written = write(fd, buffer, size);
    close(fd);

    if (bytes_written != (ssize_t)size) {
        elf_error("write_file: incomplete write");
        return -1;
    }
```

```c
    return 0;
}

/* Validate ELF file magic number and basic structure */
int is_valid_elf(unsigned char* buffer, size_t size) {
    if (!buffer || size < EI_NIDENT + sizeof(Elf64_Ehdr)) {
        return 0;
    }

    /* Check ELF magic number */
    if (memcmp(buffer, ELF_MAGIC, 4) != 0) {
        return 0;
    }

    /* Check ELF class (64-bit) */
    if (buffer[4] != 2) {  /* ELFCLASS64 */
        return 0;
    }

    /* Check data encoding (little endian) */
    if (buffer[5] != 1) {  /* ELFDATA2LSB */
        return 0;
    }

    /* Check ELF version */
    if (buffer[6] != 1) {  /* EV_CURRENT */
        return 0;
    }

    return 1;
}

/* src/common/utils.h */
#ifndef UTILS_H
#define UTILS_H

#include <stddef.h>
```

```c
unsigned char* read_file(const char* filename, size_t* size);

int write_file(const char* filename, unsigned char* buffer, size_t size);

int is_valid_elf(unsigned char* buffer, size_t size);


#endif /* UTILS_H */
```

D. Core Logic Skeleton Code

For the main CLI tool that orchestrates all milestones, here's a skeleton showing how the goals translate to implementation structure:

```c
/* tools/elflex-cli.c - Skeleton with TODOs for all milestones */

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "../src/elf/elf_loader.h"

#include "../src/patch/patcher.h"

#include "../src/trampoline/trampoline.h"

#include "../src/inject/injector.h"

#include "../src/runtime/ptrace_patcher.h"


/* Milestone 1: Instruction-level patching command */

static int handle_patch_command(int argc, char** argv) {

    if (argc < 4) {

        fprintf(stderr, "Usage: %s patch <binary> <offset> <hex_bytes>\n", argv[0]);

        return 1;

    }



    // TODO 1: Load the ELF binary using elf_load()

    // TODO 2: Parse the offset (hexadecimal string to integer)

    // TODO 3: Parse the hex bytes to replace at that offset

    // TODO 4: Validate offset is within .text section bounds

    // TODO 5: Plan the patch using plan_patch() (check size constraints)

    // TODO 6: Apply the patch using apply_patch()

    // TODO 7: Write modified binary to new file

    // TODO 8: Clean up resources with elf_binary_free()



    return 0;

}


/* Milestone 2: Function hooking with trampolines command */

static int handle_hook_command(int argc, char** argv) {

    if (argc < 4) {

        fprintf(stderr, "Usage: %s hook <binary> <function> <hook_func>\n", argv[0]);

        return 1;

    }
```

```
    // TODO 1: Load the ELF binary
    // TODO 2: Find function address using elf_get_symbol_address()
    // TODO 3: Generate trampoline code using generate_trampoline()
    // TODO 4: Relocate original prologue using relocate_prologue()
    // TODO 5: Apply fixups for RIP-relative instructions
    // TODO 6: Write trampoline and modified binary
    // TODO 7: Clean up resources

    return 0;
}

/* Milestone 3: Code injection command */
static int handle_inject_command(int argc, char** argv) {
    if (argc < 4) {
        fprintf(stderr, "Usage: %s inject <binary> <code_file> <section_name>\n", argv[0]);
        return 1;
    }

    // TODO 1: Load target ELF binary
    // TODO 2: Read code to inject from file
    // TODO 3: Create new section using inject_section()
    // TODO 4: Update ELF headers to accommodate new section
    // TODO 5: Resolve symbols from target in injected code using resolve_symbol_in_target()
    // TODO 6: Write modified binary
    // TODO 7: Clean up resources

    return 0;
}

/* Milestone 4: Runtime patching command */
static int handle_runtime_command(int argc, char** argv) {
    if (argc < 4) {
        fprintf(stderr, "Usage: %s runtime <pid> <function> <hook_func>\n", argv[0]);
        return 1;
    }
```

```
    // TODO 1: Parse PID from arguments
    // TODO 2: Attach to process using attach_to_process()
    // TODO 3: Read process memory to locate function
    // TODO 4: Generate trampoline code (similar to static hooking but in memory)
    // TODO 5: Make code pages writable using make_page_writable()
    // TODO 6: Write trampoline to process memory using write_process_memory()
    // TODO 7: Restore page protections
    // TODO 8: Flush instruction cache
    // TODO 9: Detach from process
    // TODO 10: Clean up resources

    return 0;
}

int main(int argc, char** argv) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <command> [options]\n", argv[0]);
        fprintf(stderr, "Commands:\n");
        fprintf(stderr, "  patch    - Instruction-level patching (Milestone 1)\n");
        fprintf(stderr, "  hook     - Function hooking with trampolines (Milestone 2)\n");
        fprintf(stderr, "  inject   - Code injection (Milestone 3)\n");
        fprintf(stderr, "  runtime  - Runtime patching via ptrace (Milestone 4)\n");
        return 1;
    }

    const char* command = argv[1];

    if (strcmp(command, "patch") == 0) {
        return handle_patch_command(argc - 1, argv + 1);
    } else if (strcmp(command, "hook") == 0) {
        return handle_hook_command(argc - 1, argv + 1);
    } else if (strcmp(command, "inject") == 0) {
        return handle_inject_command(argc - 1, argv + 1);
    } else if (strcmp(command, "runtime") == 0) {
        return handle_runtime_command(argc - 1, argv + 1);
    } else {
```

```
        fprintf(stderr, "Unknown command: %s\n", command);
        return 1;
    }
}
```

E. Language-Specific Hints

For C implementation on Linux:

1. **ELF Structures**: Use the standard `<elf.h>` header which provides `Elf64_Ehdr`, `Elf64_Shdr`, etc. However, be aware of potential differences across distributions.
2. **System Calls**: For ptrace operations, include `<sys/ptrace.h>` and `<sys/user.h>` for register structures.
3. **Memory Protection**: Use `syscall(SYS_mprotect, ...)` or inject a small shellcode to call `mprotect` when modifying runtime process memory.
4. **Instruction Cache**: On x86-64, the instruction cache is generally coherent, but use `__builtin___clear_cache(start, end)` from GCC/Clang for portability.
5. **Position-Independent Code**: When injecting code, compile with `-fPIC -pie` and extract the `.text` section from the resulting object file.
6. **Error Handling**: Use `perror()` for system call errors and provide custom messages for ELF parsing errors.

F. Milestone Checkpoint

After implementing the goals for each milestone, verify as follows:

Milestone 1 Checkpoint:

```bash
# Create a test binary
echo -e '#include <stdio.h>\nint main() { printf("Hello\\n"); return 0; }' > test.c
gcc -o test test.c

# Patch a NOP sled at an offset
./elflex-cli patch test 0x1150 "9090909090"  # Replace 5 bytes with NOPs

# Verify patch
objdump -d test.patched | grep -A5 "<main>:"  # Should show NOPs at patched location

# Run patched binary
./test.patched  # Should still work (unless you patched critical code)
```

Milestone 2 Checkpoint:

```bash
# Hook printf in the test binary
./elflex-cli hook test printf my_hook

# Create a simple hook library
echo -e 'void my_hook() { printf("Hooked!\\n"); }' > hook.c
gcc -fPIC -shared -o libhook.so hook.c

# Test hooked binary (requires LD_PRELOAD for now)
LD_PRELOAD=./libhook.so ./test.hooked  # Should print "Hooked!" before "Hello"
```

Milestone 3 Checkpoint:

```bash
# Compile code to inject
echo -e 'void injected() { printf("Injected!\\n"); }' > inject.c
gcc -fPIC -c inject.c -o inject.o

# Extract .text section
objcopy -O binary --only-section=.text inject.o inject.bin

# Inject into binary
./elflex-cli inject test inject.bin .inject

# Verify new section
readelf -S test.injected | grep .inject  # Should show new section
```

Milestone 4 Checkpoint:

```bash
# Start test program in background
./test &
PID=$!

# Apply runtime hook
./elflex-cli runtime $PID printf my_hook

# The running process should now print "Hooked!" before "Hello"
kill $PID  # Clean up
```

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Segmentation fault after patching	Invalid offset or corrupt ELF structure	Use `readelf -l` to check program headers, `objdump -d` to verify code location	Ensure offset is within `.text` section bounds
Hook not called	Trampoline jump not installed correctly	Disassemble patched binary at function start, check for jump instruction	Verify trampoline generation produces correct relative/absolute jump
Process hangs after ptrace attach	Process stopped but not resumed	Check ptrace workflow: attach → stop → modify → resume → detach	Ensure `PTRACE_CONT` is called after modifications
Injected code crashes	Missing relocations or wrong addressing mode	Examine injected code with `objdump -d`, check for absolute addresses	Use RIP-relative addressing exclusively or add proper relocations
Permission denied on ptrace	Insufficient privileges or YAMA restrictions	Check `cat /proc/sys/kernel/yama/ptrace_scope`, try as root	Run as root or adjust ptrace_scope (for learning only)
Modified code not executed	Instruction cache not flushed	Check if `__builtin___clear_cache` was called	Call cache flush after writing to code pages

High-Level Architecture

> **Milestone(s):** 1 (Instruction-level Patching), 2 (Function Hooking with Trampolines), 3 (Code Injection), 4 (Runtime Patching via ptrace)

The high-level architecture of ELFlex defines how the system's components collaborate to achieve the instrumentation goals across all four milestones. At its core, ELFlex follows a pipeline architecture: raw binary data flows through a series of processing stages, each responsible for a specific transformation, ultimately producing an instrumented binary or affecting a running process. This section provides a bird's-eye view of the five primary components, their responsibilities, interactions, and how they are organized in the codebase.

Architectural Components and Flow

Mental Model: The Instrumentation Factory Assembly Line

Imagine ELFlex as a specialized factory assembly line for modifying complex machinery (ELF binaries). The factory receives a piece of machinery (the target binary) and follows a series of stations:

1. **The Receiving & Inspection Station (ELF Loader)**: Unpacks the machinery, reads its blueprint (ELF headers), and creates a detailed, editable model of its internal structure.
2. **The Modification Planning Station (Patch Planner)**: Examines the model and marks exactly where changes should be made, planning cuts and additions.
3. **The Component Fabrication Station (Trampoline Generator & Code Injector)**: Builds new parts (trampolines, hook code) that will be inserted into the machinery.
4. **The Assembly & Welding Station (Binary Writer)**: Integrates the new parts into the machinery's model, adjusting the blueprint to accommodate them.
5. **The Live Retrofit Team (Runtime Patcher)**: For machinery already in operation, this team temporarily stops it, makes the planned modifications directly in place, and restarts it.

Each station specializes in one task and passes its output to the next, ensuring a clean separation of concerns. The following component diagram illustrates this data flow and dependency structure:

Architecture Overview: ELFlex modifies ELF executables to inject monitoring hooks without breaking structure or execution flow.

System Component Overview

System Component Overview

Trampoline-based binary instrumentation for ELF executables

ELF Loader

- Parses ELF executable structure
- Extracts sections and segments
- Resolves symbols and relocation info

Patch Planner

- Analyzes function entry/exit points
- Determines hook placement strategy
- Calculates trampoline insertion locations

Trampoline Generator

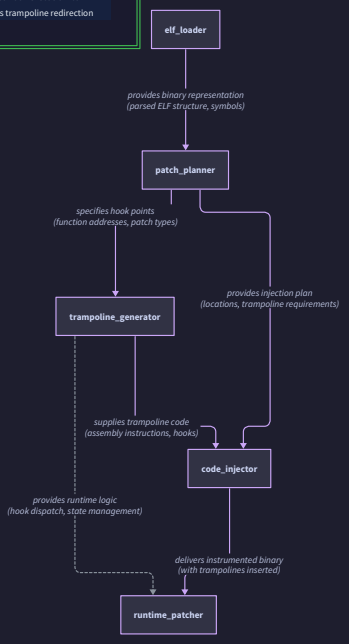
- Creates trampoline assembly code
- Handles instruction preservation
- Generates hook redirection logic

Code Injector

- Inserts trampolines into binary
- Updates ELF metadata
- Manages code cave allocation

Runtime Patcher

- Applies dynamic patches at runtime
- Manages hook execution flow
- Handles trampoline redirection



Component Descriptions

Component	Primary Milestone(s)	Key Responsibilities	Key Data Structures Produced/Consumed	Dependencies
ELF Loader	1 (foundation for all)	Load ELF file from disk into memory, parse headers and sections, validate ELF structure, provide query interface for sections and symbols.	`elf_binary_t`, `section_t`, `segment_t`	File I/O utilities
Patch Planner	1, 2	Plan instruction-level modifications, validate patch locations within executable sections, generate padding (NOP sleds) for size mismatches, manage offset adjustments.	`patch_site_t` (planned patch metadata)	ELF Loader (for binary model)
Trampoline Generator	2	Generate position-independent trampoline code sequences, relocate original function prologues, fix RIP-relative addressing in relocated code, manage register save/restore.	`trampoline_t` (code + metadata)	ELF Loader (for symbol addresses), Patch Planner (to plan prologue replacement)
Code Injector	3	Allocate space for new code (via new section or code cave), add/modify ELF section and program headers, resolve symbol references between injected and original code, apply relocations.	`injected_section_t`	ELF Loader (for binary model), Trampoline Generator (for trampoline code)
Binary Writer	1, 2, 3	Serialize the modified in-memory ELF model back to a valid ELF file on disk, handling all offset recalculations, header updates, and structural integrity.	(Consumes `elf_binary_t`)	All components that modify the binary model
Runtime Patcher	4	Attach to a running process via ptrace, read/write process memory, modify memory protections, inject code and trampolines directly into the process's address space.	Runtime session state	ELF Loader (for analysis), Trampoline Generator (for code)

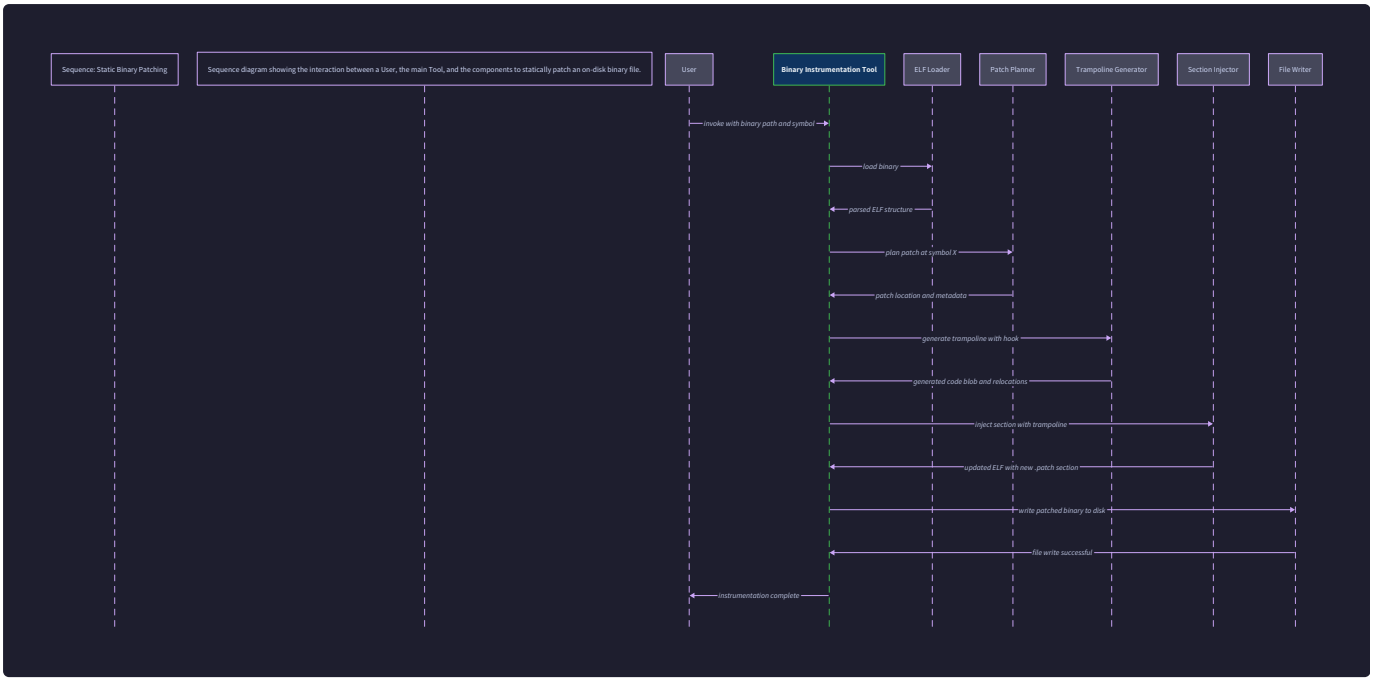
End-to-End Data Flow

The interaction between components follows two primary workflows: **static instrumentation** (for on-disk binaries) and **runtime instrumentation** (for running processes).

Static Instrumentation Flow (Milestones 1-3): This flow patches an ELF file on disk and writes a new, instrumented binary.

1. **Load**: The `elf_load` function reads the target binary file, parses its ELF structure, and constructs an in-memory `elf_binary_t` model.
2. **Analyze & Plan**: Based on user requests (e.g., "hook function `foo`"), the tool queries the model via `elf_get_symbol_address` to locate target functions. The Patch Planner then creates a `patch_site_t` for each modification point.
3. **Generate Code**: For function hooks, the Trampoline Generator creates a `trampoline_t` containing the redirect code. If custom hook functions need injection, the Code Injector prepares an `injected_section_t` with the new code and updates the binary model.
4. **Apply Patches**: The Patch Planner applies each `patch_site_t` to the binary model, replacing instructions in the `.text` section with jumps to trampolines or other code.
5. **Write Output**: The Binary Writer traverses the updated `elf_binary_t` model, recalculates all file offsets, and writes a new, valid ELF file to disk.

This sequence is visualized in the static patching diagram:



Runtime Instrumentation Flow (Milestone 4): This flow modifies a currently executing process without writing a new file.

1. **Attach**: The Runtime Patcher calls `attach_to_process` to attach to the target process via ptrace, stopping its execution.
2. **Analyze**: The ELF Loader may analyze the process's executable file (from `/proc/pid/exe`) to locate symbol addresses, or the tool may use pre-calculated offsets.
3. **Generate Code**: Similar to static flow, the Trampoline Generator creates trampoline code. However, this code must be position-independent and ready for injection at a specific memory address.
4. **Memory Modification**: The Runtime Patcher uses `make_page_writable` to change memory protections on the target's code pages, then `write_process_memory` to inject the trampoline code into the process's address space (either into an existing code cave or a newly allocated memory region). It then patches the target function in memory by writing a jump instruction over its prologue.
5. **Detach**: After flushing the instruction cache, the Runtime Patcher detaches from the process, allowing it to resume execution with the hook active.

> **Key Architectural Insight:** The separation between static and runtime instrumentation is largely at the I/O boundary. The core logic for analyzing binaries, planning patches, and generating trampolines is shared. The Runtime Patcher component essentially replaces the Binary Writer, performing modifications directly in memory rather than on a file.

Architecture Decision Record: Monolithic Tool vs. Plugin Architecture

> **Decision: Integrated Monolithic Tool over Plugin Framework**
>
> - **Context**: The tool needs to perform a sequence of complex, interdependent operations (loading, analysis, patching, writing). We must choose between building a single integrated program or a framework where each component is a separate plugin.
> - **Options Considered**:
>   1. **Monolithic Tool**: A single executable with well-separated internal modules but a unified codebase and control flow.
>   2. **Plugin Framework**: A core engine that loads plugins for each stage (loader, patcher, injector), allowing third-party extensions.
> - **Decision**: Build as a monolithic tool with clean internal module boundaries.
> - **Rationale**: The primary goal is learning the fundamentals of binary instrumentation, not building an extensible platform. A monolithic design reduces complexity (no plugin loading, versioning, or IPC overhead), simplifies debugging, and ensures tight integration between components that need to share complex in-memory state (like the `elf_binary_t` model). The learning focus is on the algorithms and low-level operations, not on framework design.
> - **Consequences**: The tool will be less flexible for future extension (adding support for new architectures or patch strategies requires recompilation). However, the clear module separation (via directory structure and header files) allows internal components to be developed, tested, and understood independently. This aligns with the educational objective.

Option	Pros	Cons	Chosen?
Monolithic Tool	Simpler build/debug, direct data sharing, no IPC overhead, faster execution	Less extensible, components coupled at compile-time	**Yes** (educational focus)
Plugin Framework	Highly extensible, components can be swapped independently, supports third-party contributions	Significant complexity overhead, IPC serialization for shared data, harder to debug	No

Recommended File/Module Structure

Organizing the codebase with clear module boundaries is critical for managing complexity and aligning with the architectural components. The following directory and file structure separates concerns, promotes reusability, and mirrors the component diagram.

Mental Model: The Workshop Layout

Think of the project directory as a workshop where each bench (directory) holds tools and materials for a specific task. The main workshop area (`src/`) contains benches for each major component, while a shared tools wall (`include/` and `common/`) holds utilities used by multiple benches. The entry point (`elflex.c`) is the foreman's desk where work orders are received and dispatched.

Directory Structure

```
elflex/                    # Project root
├── CMakeLists.txt         # Build configuration (or Makefile)
├── README.md
├── docs/                  # Design documents, diagrams
├── include/               # Public header files (API exposed to users)
│   ├── elflex.h           # Main public API for the tool
│   └── elf_abi.h          # ELF structure definitions (Elf64_Ehdr, etc.)
├── src/                   # Source code
│   ├── main/
│   │   └── elflex.c       # Command-line interface, main() entry point
│   ├── loader/            # ELF Loader component
│   │   ├── elf_loader.c
│   │   ├── elf_loader.h   # Internal header for loader module
│   │   └── elf_parser.c   # Low-level ELF parsing utilities
│   ├── patch/             # Patch Planner component
│   │   ├── patch_planner.c
│   │   ├── patch_planner.h
│   │   └── nop_generator.c
│   ├── trampoline/        # Trampoline Generator component
│   │   ├── trampoline_gen.c
│   │   ├── trampoline_gen.h
│   │   └── relocator.c    # Prologue relocation & RIP fixup logic
│   ├── inject/            # Code Injector component
│   │   ├── code_injector.c
│   │   ├── code_injector.h
│   │   └── symbol_resolver.c
│   ├── writer/            # Binary Writer component
│   │   ├── binary_writer.c
│   │   └── binary_writer.h
│   ├── runtime/           # Runtime Patcher component
│   │   ├── runtime_patcher.c
│   │   ├── runtime_patcher.h
│   │   └── ptrace_utils.c  # Helpers for ptrace operations
│   └── common/            # Shared utilities
│       ├── util.c         # read_file, write_file, etc.
│       ├── util.h
│       ├── logging.c      # Logging macros and functions
│       └── logging.h
└── tests/                 # Test suites
    ├── unit/              # Unit tests per component
    └── integration/       # End-to-end tests with sample binaries
```

Module Interdependencies

The dependency graph between modules should be a directed acyclic graph to avoid circular dependencies. The following table summarizes the allowed inclusion relationships:

Module	May Include Headers From	Rationale
`main/`	All component headers, `include/elflex.h`	Coordinates the overall workflow
`loader/`	`common/`, `include/elf_abi.h`	Pure ELF parsing, independent of patching logic
`patch/`	`loader/`, `common/`	Needs binary model to plan patches
`trampoline/`	`loader/`, `patch/`, `common/`	Uses symbol addresses, may create patch sites
`inject/`	`loader/`, `trampoline/`, `common/`	Injects trampolines, needs symbol resolution
`writer/`	`loader/`, `common/`	Serializes the binary model, independent of patch origin
`runtime/`	`loader/`, `trampoline/`, `common/`	Uses analysis and code generation, but not disk writer

Note that `patch/`, `trampoline/`, and `inject/` are allowed to depend on `loader/` because they operate on the binary model. However, `loader/` should not depend on them, preserving the unidirectional flow of data.

Header File Design

Each component directory contains a header file (e.g., `elf_loader.h`) that declares the public interface for that component. These headers should follow the principle of minimal exposure: only declare functions and structures that other components need to use. Internal helper functions and structures should be declared statically within the `.c` file or in a private header not included elsewhere.

The `include/elflex.h` public API provides a simplified, high-level interface for users of the library (if exposed as a library) or for the command-line tool. It might wrap complex sequences like "hook function X with hook Y" into a single function call that internally orchestrates multiple components.

Build System Considerations

A build system like CMake or a well-structured Makefile should compile each component into its own object file, then link them together into the final `elflex` executable. This modular compilation ensures that changes to one component don't require recompiling others, speeding up development.

> **Key Architectural Insight:** This file structure enforces the architectural separation of concerns. If a developer finds themselves needing to include `runtime_patcher.h` in `elf_loader.c`, it's a signal that they're likely mixing responsibilities and should reconsider the design. The directory layout acts as a physical manifestation of the component boundaries.

Common Pitfalls in Architectural Design

⚠️ Pitfall: Tight Coupling Between Components via Global State

- **Description**: Storing the `elf_binary_t` model or other shared state in global variables accessible by all components, rather than passing it explicitly as function parameters.
- **Why It's Wrong**: This creates hidden dependencies, makes testing difficult (components can't be isolated), and leads to concurrency issues if the tool is ever extended to multi-threading. It violates the principle of explicit data flow.
- **Fix**: Design component interfaces to accept all necessary state as parameters. For example, `apply_patch` should take a `elf_binary_t*` argument, not rely on a global variable. This makes data flow visible and components reusable.

⚠️ Pitfall: Mixing ELF Parsing Logic with Instrumentation Logic

- **Description**: Putting code that analyzes function prologues or plans trampolines directly into the ELF loader module.
- **Why It's Wrong**: The loader's responsibility is to provide an accurate model of the binary, not to interpret its contents. This mixing makes the loader complex and harder to test, and it violates the single responsibility principle.
- **Fix**: Keep the loader focused on parsing ELF structures. Have separate components (Patch Planner, Trampoline Generator) that use the loader's query functions to get data, then perform analysis and planning.

⚠️ Pitfall: Inconsistent Error Handling Across Components

- **Description**: Some components return error codes, others log and exit, others return NULL on failure without context.
- **Why It's Wrong**: Makes the tool fragile and hard to debug. Users get inconsistent behavior, and errors might be silently ignored as they propagate.
- **Fix**: Define a consistent error handling strategy early. For ELFlex, a good approach is for all public component functions to return an `int` (0 for success, negative for error) and optionally provide error details via a thread-local or passed-in error structure. Use `elf_error` for logging but don't let it terminate the program except in fatal cases.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option
ELF Parsing	Manual parsing using `elf_abi.h` definitions	Use libelf (ELF access library) for more robust handling
Instruction Encoding	Hardcoded byte arrays for simple instructions (NOP=0x90, JMP=0xE9)	Integrate a disassembler/assembler library (e.g., Capstone, Zydis) for complex instruction generation
Code Injection	Append new section at end of file, shift all following data	Sophisticated segment gap detection, use existing padding in PT_LOAD segments
Runtime Patching	Basic ptrace with `PTRACE_PEEKDATA/POKEDATA`	Use `process_vm_readv/writev` for bulk memory operations, integrate with seccomp for safety

Recommended File Structure Implementation:

The directory structure above should be created at project start. Each component's `.c` and `.h` files should begin with a standard header comment and include only necessary dependencies. Below is a template for a component header file:

```c
// src/loader/elf_loader.h

#ifndef ELF_LOADER_H

#define ELF_LOADER_H


#include <stdint.h>

#include <stddef.h>

#include "common/error.h"  // For error handling utilities

#include "include/elf_abi.h" // For ELF type definitions


// Forward declarations of internal structures

typedef struct elf_binary_s elf_binary_t;

typedef struct section_s section_t;

typedef struct segment_s segment_t;


// Public API functions

elf_binary_t* elf_load(const char* filename, error_t* err);

section_t* elf_find_section(elf_binary_t* binary, const char* name);

uint64_t elf_get_symbol_address(elf_binary_t* binary, const char* symbol_name);

void elf_binary_free(elf_binary_t* binary);


// Accessor functions for internal structures (if needed by other components)

Elf64_Shdr* section_get_header(section_t* section);

unsigned char* section_get_data(section_t* section);


#endif // ELF_LOADER_H
```

Infrastructure Starter Code (Common Utilities):

The `common/util.c` and `common/logging.c` provide foundational utilities used across components. Here's a complete implementation for basic file I/O and logging:

```c
// src/common/util.c

#include "common/util.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>


unsigned char* read_file(const char* filename, size_t* out_size) {

    FILE* file = fopen(filename, "rb");

    if (!file) {

        perror("fopen");

        return NULL;

    }



    fseek(file, 0, SEEK_END);

    long file_size = ftell(file);

    if (file_size < 0) {

        perror("ftell");

        fclose(file);

        return NULL;

    }

    fseek(file, 0, SEEK_SET);



    unsigned char* buffer = (unsigned char*)malloc(file_size);

    if (!buffer) {

        perror("malloc");

        fclose(file);

        return NULL;

    }



    size_t bytes_read = fread(buffer, 1, file_size, file);

    if (bytes_read != (size_t)file_size) {

        perror("fread");

        free(buffer);

        fclose(file);

        return NULL;

    }
```

```
    fclose(file);
    *out_size = bytes_read;
    return buffer;
}

int write_file(const char* filename, unsigned char* buffer, size_t size) {
    FILE* file = fopen(filename, "wb");
    if (!file) {
        perror("fopen");
        return -1;
    }

    size_t bytes_written = fwrite(buffer, 1, size, file);
    fclose(file);

    if (bytes_written != size) {
        fprintf(stderr, "Failed to write entire file\n");
        return -1;
    }
    return 0;
}

int is_valid_elf(unsigned char* buffer, size_t size) {
    if (size < 4) return 0;
    return (buffer[0] == 0x7f && buffer[1] == 'E' &&
            buffer[2] == 'L' && buffer[3] == 'F');
}
```

Core Logic Skeleton for Main Orchestration:

The main entry point in `src/main/elflex.c` orchestrates the component flow. Below is a skeleton showing how components interact for static patching:

```c
// src/main/elflex.c

#include <stdio.h>

#include <stdlib.h>

#include "loader/elf_loader.h"

#include "patch/patch_planner.h"

#include "trampoline/trampoline_gen.h"

#include "inject/code_injector.h"

#include "writer/binary_writer.h"

#include "common/error.h"


int instrument_static_binary(const char* input_path, const char* output_path,

                             const char* function_to_hook, uint64_t hook_addr) {

    error_t err = {0};



    // TODO 1: Load the binary using elf_load()

    elf_binary_t* binary = elf_load(input_path, &err);

    if (!binary) {

        fprintf(stderr, "Failed to load binary: %s\n", err.message);

        return -1;

    }



    // TODO 2: Find the address of the target function using elf_get_symbol_address()

    uint64_t target_addr = elf_get_symbol_address(binary, function_to_hook);

    if (target_addr == 0) {

        fprintf(stderr, "Function '%s' not found\n", function_to_hook);

        elf_binary_free(binary);

        return -1;

    }



    // TODO 3: Generate trampoline code for hooking

    trampoline_t* trampoline = generate_trampoline(target_addr, hook_addr, &err);

    if (!trampoline) {

        fprintf(stderr, "Failed to generate trampoline: %s\n", err.message);

        elf_binary_free(binary);

        return -1;

    }
```

```
// TODO 4: Plan the patch at the function entry point
// The trampoline includes the original prologue bytes to replace
patch_site_t* patch = plan_patch(binary, target_addr,
                                 trampoline->prologue_replacement,
                                 trampoline->prologue_replacement_size,
                                 &err);
if (!patch) {
    fprintf(stderr, "Failed to plan patch: %s\n", err.message);
    trampoline_free(trampoline);
    elf_binary_free(binary);
    return -1;
}

// TODO 5: If hook function needs to be injected, allocate space and inject it
// (This step might be skipped if hook_addr is already in binary)

// TODO 6: Apply the patch to the binary model
if (apply_patch(binary, patch, &err) != 0) {
    fprintf(stderr, "Failed to apply patch: %s\n", err.message);
    patch_free(patch);
    trampoline_free(trampoline);
    elf_binary_free(binary);
    return -1;
}

// TODO 7: Write the modified binary model to disk
if (binary_write(binary, output_path, &err) != 0) {
    fprintf(stderr, "Failed to write binary: %s\n", err.message);
    patch_free(patch);
    trampoline_free(trampoline);
    elf_binary_free(binary);
    return -1;
}
```

```
    // TODO 8: Clean up resources
    patch_free(patch);
    trampoline_free(trampoline);
    elf_binary_free(binary);

    printf("Successfully instrumented binary: %s -> %s\n", input_path, output_path);
    return 0;
}

int main(int argc, char** argv) {
    // Parse command line arguments...
    // For milestone 1 & 2 demo:
    // instrument_static_binary(argv[1], argv[2], "target_function", 0x123456);
    return 0;
}
```

Language-Specific Hints (C):

- **Memory Management**: Since C lacks RAII, be meticulous about pairing every allocation with a free. Consider creating `_free()` functions for each complex structure (e.g., `elf_binary_free()`, `trampoline_free()`) that handle deep cleanup.
- **Error Propagation**: Use a consistent error structure passed by pointer to functions, allowing error messages to bubble up without losing context. Avoid using `errno` exclusively, as library functions may overwrite it.
- **Portability**: Use fixed-width integer types (`uint64_t`, `int32_t`) from `stdint.h` for fields that map directly to ELF structures, ensuring consistent behavior across architectures.
- **Alignment**: When creating new ELF sections or segments, respect alignment requirements (`sh_addralign`, `p_align`). Use the `align_up` macro: `#define ALIGN_UP(addr, align) (((addr) + (align) - 1) & ~((align) - 1))`.

Milestone Checkpoint - Architecture Validation:

After setting up the directory structure and implementing the skeleton above, verify the architecture by:

1. **Compilation Test**: Run `make` or `cmake --build .` to ensure all components compile without circular dependency errors.
2. **Module Isolation Test**: Create a minimal test program that includes only `elf_loader.h` and calls `elf_load()` on a simple binary. This should compile and link without needing other components.
3. **Data Flow Test**: Add debug prints to each component's main functions and run the tool with a simple patch request. Verify the execution order matches the sequence diagram: Loader → Planner → Generator → Writer.

Expected output for a successful architecture setup:

```
$ make
gcc -I./include -I./src -c src/loader/elf_loader.c -o build/loader/elf_loader.o
gcc -I./include -I./src -c src/common/util.c -o build/common/util.o
... (all components compile successfully)
gcc build/main/*.o build/loader/*.o ... -o elflex
$ ./elflex --help
ELFlex: A binary instrumentation tool
Usage: ./elflex [options] <input> <output>
...
```

If linking fails with "undefined reference" errors, check that function declarations in headers match definitions, and that all required components are included in the build. If you encounter include path issues, verify the `-I` flags in your build system point to the correct directories.

Data Model

> **Milestone(s):** 1 (Instruction-level Patching), 2 (Function Hooking with Trampolines), 3 (Code Injection)

The **Data Model** forms the foundational memory representation that all other components operate upon. It's the in-memory "workshop" where the binary is disassembled, analyzed, modified, and prepared for writing. A well-designed data model separates the complex concerns of ELF format manipulation from the business logic of patching and instrumentation, enabling clean interfaces and predictable state transitions.

Core Data Structures

Mental Model: The Workshop's Blueprint and Inventory Imagine you're a carpenter renovating a complex piece of furniture (the ELF binary). You need:

1. A **complete blueprint** (`elf_binary_t`) showing all parts, their positions, and how they connect.
2. **Marked-up plans** (`patch_site_t`) specifying exactly where to cut, replace, or add new pieces.
3. **Prefabricated components** (`trampoline_t`) like custom joints or brackets that redirect forces (execution flow).
4. **New parts inventory** (`injected_section_t`) for entirely new modules you're adding to the furniture.

This data model provides all these representations, ensuring every component works from a single source of truth about the binary's current state.

ELF Binary Representation (`elf_binary_t`)

This is the central structure representing an entire ELF binary loaded into memory. It maintains both the raw byte buffer and parsed structural information for efficient navigation.

Field Name	Type	Description
`buffer`	`unsigned char*`	Pointer to the complete binary image in memory (a copy of the original file plus modifications). This is the single source of truth for the binary's current content.
`size`	`size_t`	Total size of the `buffer` in bytes. This grows when sections are injected.
`ehdr`	`Elf64_Ehdr*`	Pointer to the ELF header within `buffer`. Used for quick access to global metadata (entry point, header counts, offsets).
`sections`	`section_t**`	Array of pointers to parsed section structures. Provides O(1) access by index and enables iteration.
`section_count`	`size_t`	Number of elements in the `sections` array. Matches `e_shnum` from the header.
`segments`	`segment_t**`	Array of pointers to parsed program segment (loadable) structures. Critical for understanding memory layout.
`segment_count`	`size_t`	Number of elements in the `segments` array. Matches `e_phnum` from the header.
`text_section`	`section_t*`	Direct pointer to the `.text` section (executable code). Cached for fast access during common patching operations.
`symtab`	`section_t*`	Direct pointer to the symbol table section (usually `.symtab`). Used for function name to address resolution.

> **Design Insight:** Maintaining a raw `buffer` alongside parsed structures allows low-level byte manipulation (patching) while preserving high-level semantic navigation (finding functions). The cached `text_section` and `symtab` pointers optimize the common case where most patches target code symbols.

Section Representation (`section_t`)

Represents a single ELF section with its header and actual data. Sections are logical divisions (code, data, debug info) within the binary.

Field Name	Type	Description
`shdr`	`Elf64_Shdr*`	Pointer to the section header within `buffer`. Contains metadata like type, flags, virtual address, file offset, and size.
`name`	`char*`	Null-terminated string of the section name (e.g., ".text", ".data"). Retrieved from the string table section.
`data`	`unsigned char*`	Pointer to the section's content within `buffer`. This is a direct pointer into the `buffer` for efficient in-place modification.

Segment Representation (`segment_t`)

Represents a loadable program segment that the operating system loads into memory. Segments define runtime memory regions and permissions.

Field Name	Type	Description
`phdr`	`Elf64_Phdr*`	Pointer to the program header within `buffer`. Defines memory permissions (`p_flags`), file-to-memory mapping (`p_offset` , `p_vaddr`), and size (`p_filesz` , `p_memsz`).
`data`	`unsigned char*`	Pointer to the segment's content within `buffer`. For `PT_LOAD` segments, this corresponds to the initial memory image.

Standard ELF Header Structures

These are standard ELF64 definitions as per the specification. We include them for completeness in the data model.

`Elf64_Ehdr` (ELF Header)

Field Name	Type	Description
`e_ident`	`unsigned char[16]`	Magic number and identification info (ELF class, data encoding, version).
`e_type`	`Elf64_Half`	Object file type (executable, shared object, core, etc.).
`e_machine`	`Elf64_Half`	Target architecture (x86-64 = 0x3E).
`e_version`	`Elf64_Word`	ELF version (always 1 for current).
`e_entry`	`Elf64_Addr`	Virtual address of the program entry point.
`e_phoff`	`Elf64_Off`	File offset to the start of the program header table.
`e_shoff`	`Elf64_Off`	File offset to the start of the section header table.
`e_flags`	`Elf64_Word`	Processor-specific flags.
`e_ehsize`	`Elf64_Half`	Size of this ELF header in bytes.
`e_phentsize`	`Elf64_Half`	Size of one program header table entry.
`e_phnum`	`Elf64_Half`	Number of entries in the program header table.
`e_shentsize`	`Elf64_Half`	Size of one section header table entry.
`e_shnum`	`Elf64_Half`	Number of entries in the section header table.
`e_shstrndx`	`Elf64_Half`	Section header table index of the section name string table.

`Elf64_Shdr` (Section Header)

Field Name	Type	Description
`sh_name`	`Elf64_Word`	Index into the section name string table.
`sh_type`	`Elf64_Word`	Section type (`SHT_PROGBITS` , `SHT_SYMTAB` , etc.).
`sh_flags`	`Elf64_Xword`	Section attributes (`SHF_WRITE` , `SHF_ALLOC` , `SHF_EXECINSTR`).
`sh_addr`	`Elf64_Addr`	Virtual address of the section in memory (0 if not loadable).
`sh_offset`	`Elf64_Off`	File offset to the section's data.
`sh_size`	`Elf64_Xword`	Size of the section in bytes.
`sh_link`	`Elf64_Word`	Section header table index link (interpretation depends on `sh_type`).
`sh_info`	`Elf64_Word`	Extra information (interpretation depends on `sh_type`).
`sh_addralign`	`Elf64_Xword`	Section alignment constraint (power of two).
`sh_entsize`	`Elf64_Xword`	Size of each entry for sections containing fixed-size entries (e.g., symbol table).

`Elf64_Phdr` (Program Header)

Field Name	Type	Description
`p_type`	`Elf64_Word`	Segment type (`PT_LOAD` , `PT_DYNAMIC` , etc.).
`p_flags`	`Elf64_Word`	Segment flags (`PF_R` , `PF_W` , `PF_X`).
`p_offset`	`Elf64_Off`	File offset to the segment data.
`p_vaddr`	`Elf64_Addr`	Virtual address of the segment in memory.
`p_paddr`	`Elf64_Addr`	Physical address (unspecified for most systems, usually same as `p_vaddr`).
`p_filesz`	`Elf64_Xword`	Size of the segment in the file.
`p_memsz`	`Elf64_Xword`	Size of the segment in memory (may be larger for `.bss`).
`p_align`	`Elf64_Xword`	Segment alignment in memory and file.

`Elf64_Sym` (Symbol Table Entry)

Field Name	Type	Description
`st_name`	`Elf64_Word`	Index into the symbol name string table.
`st_info`	`unsigned char`	Symbol type and binding attributes.
`st_other`	`unsigned char`	Symbol visibility.
`st_shndx`	`Elf64_Half`	Section index containing the symbol (or special values like `SHN_ABS`).
`st_value`	`Elf64_Addr`	Symbol value (address for functions/data).
`st_size`	`Elf64_Xword`	Symbol size in bytes.

Patch Metadata (`patch_site_t`)

Represents a planned modification at a specific location in the binary. It's a "work order" that specifies what to change, where, and how to handle size mismatches.

Field Name	Type	Description
`target_offset`	`uint64_t`	File offset within the binary `buffer` where the patch will be applied.
`original_size`	`size_t`	Size in bytes of the region to be replaced.
`new_bytes`	`unsigned char*`	Pointer to the new byte sequence to insert.
`new_size`	`size_t`	Size of the `new_bytes` array.
`nop_padding`	`unsigned char*`	Pointer to a NOP sled (if `new_size < original_size`) to fill the remaining bytes.
`nop_padding_size`	`size_t`	Size of the NOP sled (`original_size - new_size` if padding needed).
`applied`	`int`	Boolean flag indicating whether this patch has been applied to the in-memory `buffer`.

ADR: Patch Representation as Separate Metadata

- **Context:** When patching, we need to track what changes are planned versus applied, support undo/validation, and handle size mismatches.
- **Options Considered:**
  1. **Direct in-place modification:** Immediately write new bytes to `buffer` when user requests patch.
  2. **Patch list with metadata:** Maintain a list of planned patches that can be validated and applied atomically.
- **Decision:** Patch list with metadata (`patch_site_t`).
- **Rationale:** Separating planning from application allows validation of all patches before modifying the binary (e.g., checking for overlapping patches). It also enables features like dry-run, patch rollback, and batch application. The metadata tracks padding requirements cleanly.
- **Consequences:** Adds memory overhead for patch metadata and requires explicit apply step. However, this yields a more robust and debuggable system.

Trampoline Representation (`trampoline_t`)

Represents a generated trampoline: the actual machine code and metadata needed to hook a function.

Field Name	Type	Description
`code`	`unsigned char*`	Pointer to the generated trampoline machine code bytes.
`code_size`	`size_t`	Size of the trampoline code in bytes.
`original_prologue`	`unsigned char*`	Copy of the original function's prologue instructions that were displaced.
`prologue_size`	`size_t`	Size of the displaced prologue in bytes.
`relocated_prologue`	`unsigned char*`	Pointer to the relocated prologue with RIP-relative fixups applied.
`relocated_size`	`size_t`	Size of the relocated prologue (may be larger due to fixup expansion).
`hook_address`	`uint64_t`	Virtual address where the user's hook function will be placed (or is already located).
`target_function_address`	`uint64_t`	Virtual address of the original function being hooked.
`trampoline_address`	`uint64_t`	Virtual address where this trampoline will be placed in the binary (injected section).

Injected Section Representation (`injected_section_t`)

Represents a new section being added to the binary, typically containing hook functions or trampolines.

Field Name	Type	Description
`name`	`char*`	Name of the new section (e.g., ".inject").
`data`	`unsigned char*`	Pointer to the raw content of the section (position-independent code).
`data_size`	`size_t`	Size of the section content in bytes.
`vaddr`	`uint64_t`	Assigned virtual address for this section (aligned appropriately).
`shdr`	`Elf64_Shdr*`	Pointer to the newly created section header within the updated `buffer`.
`requires_relocation`	`int`	Boolean flag indicating whether this section contains relocations that need processing.
`symbol_dependencies`	`char**`	Array of symbol names that need resolution within the target binary.
`dependency_count`	`size_t`	Number of symbol dependencies.

Error Handling Structure (`error_t`)

A unified structure for propagating detailed error information across components.

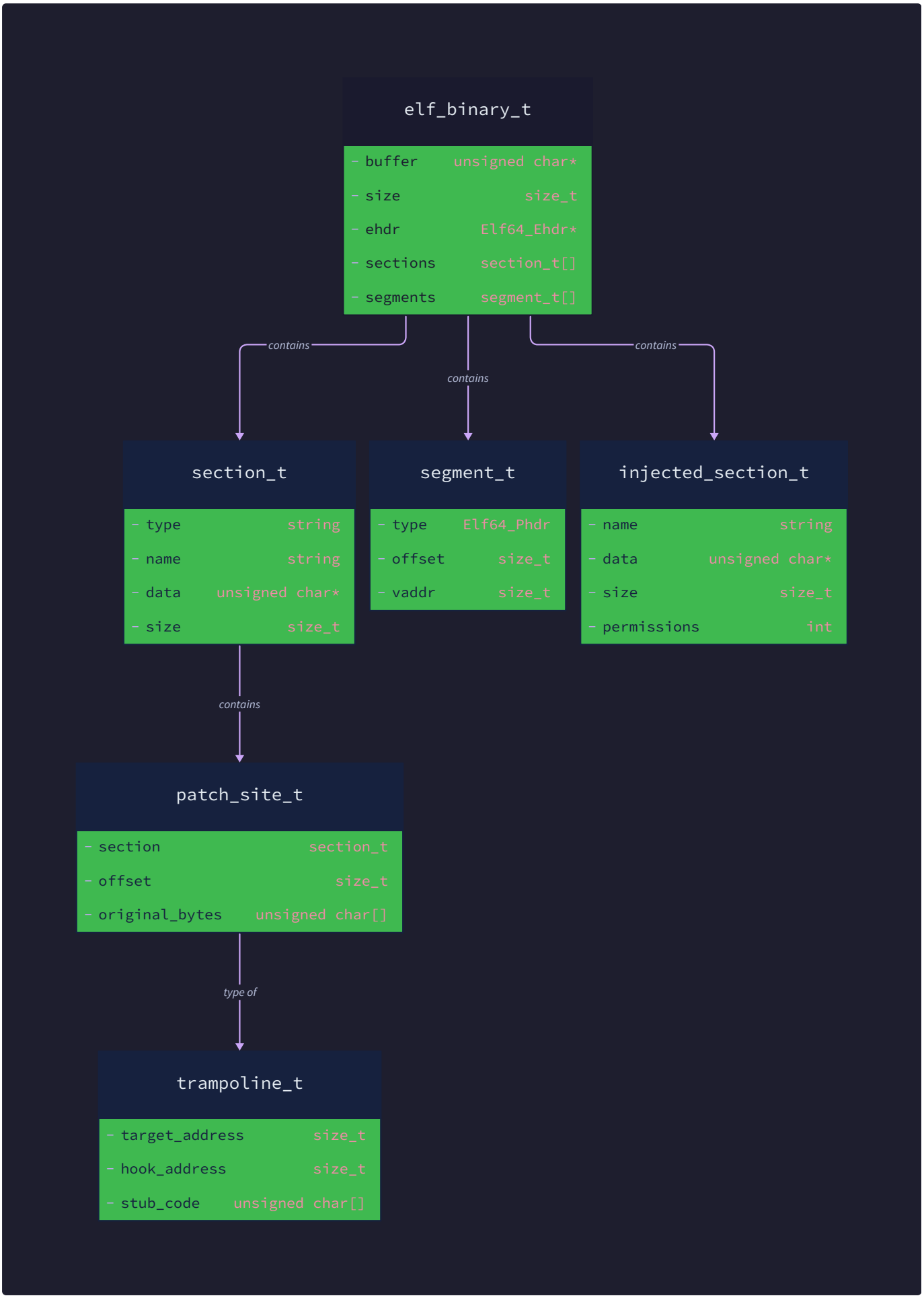
Field Name	Type	Description
`code`	`int`	Numeric error code (e.g., `ERR_INVALID_ELF`, `ERR_PATCH_OVERLAP`).
`message`	`char[256]`	Human-readable error description.
`context`	`char[128]`	Additional context about where the error occurred (function name, offset, etc.).

Relationships and State

Mental Model: The Workshop's Assembly Line The data structures interact in a precise pipeline that mirrors the physical flow of a manufacturing line:

1. **Raw Material Intake:** `elf_binary_t` is created from a file, parsing all sections and segments.
2. **Design Station:** `patch_site_t` and `trampoline_t` are planned based on user requests and symbol lookup.
3. **Fabrication Station:** `injected_section_t` is prepared with new code, resolving symbol dependencies.
4. **Assembly Station:** All modifications are applied to the `elf_binary_t`'s `buffer`.
5. **Quality Control:** The modified `elf_binary_t` is validated before being written to disk.

The following diagram illustrates these relationships:


elf_binary_t
- buffer unsigned char*
- size size_t
- ehdr Elf64_Ehdr*
- sections section_t[]
- segments segment_t[]
contains
contains
contains
section_t
- type string
- name string
- data unsigned char*
- size size_t
segment_t
- type Elf64_Phdr
- offset size_t
- vaddr size_t
injected_section_t
- name string
- data unsigned char*
- size size_t
- permissions int
contains
patch_site_t
- section section_t
- offset size_t
- original_bytes unsigned char[]
type of
trampoline_t
- target_address size_t
- hook_address size_t
- stub_code unsigned char[]

Structural Relationships

- **Containment:** `elf_binary_t` **contains** arrays of `section_t` and `segment_t` structures. Each `section_t` and `segment_t` holds pointers **into** the `buffer` rather than owning separate memory, ensuring consistency.
- **Reference:** `patch_site_t` **references** a location within the `buffer` via `target_offset`. It does not store a pointer directly to the buffer because offsets remain valid even if the buffer is reallocated during injection.
- **Specialization:** `trampoline_t` is a specialized type of code patch that includes metadata specific to function hooking (prologue handling, hook addresses). It may generate a corresponding `patch_site_t` for inserting the trampoline jump at the function entry.
- **Aggregation:** `injected_section_t` **aggregates** code and data that will become a new section in the `elf_binary_t`. It may contain multiple `trampoline_t` structures and hook functions.

State Lifecycle of a Patch Operation

The data model goes through a well-defined state sequence during a typical instrumentation operation:

1. **Initial Loaded State:**
   - `elf_binary_t` created with `elf_load()`
   - `buffer` contains exact copy of input file
   - All section/segment pointers valid within `buffer`
   - No patches or injections present
2. **Planning State:**
   - User requests hook on function `foo`
   - `elf_get_symbol_address()` returns virtual address of `foo`
   - `generate_trampoline()` creates a `trampoline_t` with:
     - Original prologue copied from `buffer` at function address
     - Relocated prologue generated with fixups
     - Trampoline code generated (save registers, call hook, etc.)
   - `plan_patch()` creates a `patch_site_t` for the function entry:
     - `target_offset` = virtual address to file offset conversion
     - `new_bytes` = jump instruction to trampoline location
     - `original_size` = size of displaced prologue
     - Padding calculated if jump instruction is smaller than prologue
3. **Injection Preparation State:**
   - `inject_section()` creates an `injected_section_t` for trampolines
   - Section data assembled from one or more `trampoline_t.code` blocks
   - `resolve_symbol_in_target()` fills in addresses for dependencies
   - `allocate_code_cave()` determines where to place the new section
4. **Modification State:**
   - `apply_patch()` writes the jump instruction (and NOP padding) to `buffer` at `target_offset`
   - `buffer` may be reallocated and expanded if new section is added
   - Section and program header tables updated in `buffer`
   - All internal pointers (`section_t.data`, `segment_t.data`) remain valid because they point into `buffer`
   - Patch `applied` flag set to true
5. **Final State (Ready for Write):**
   - All planned patches applied
   - New sections integrated
   - `elf_binary_t` represents a complete, valid ELF binary
   - `binary_write()` can serialize the `buffer` to disk

Key Invariants

To maintain consistency throughout state transitions, the following invariants must always hold:

1. **Buffer Consistency:** All `section_t.data` and `segment_t.data` pointers must point to valid locations within `elf_binary_t.buffer` for their entire claimed size.
2. **Offset Validity:** For any virtual address `vaddr` in a loadable segment, the corresponding file offset `offset = vaddr - p_vaddr + p_offset` must be within `buffer`.
3. **Patch Non-Overlap:** No two `patch_site_t` structures may have overlapping `target_offset` ranges (`[target_offset, target_offset + original_size)`).
4. **Header Integrity:** The `ehdr` must correctly reflect the number of sections and segments after injection (`e_shnum`, `e_phnum`).
5. **Alignment Preservation:** All section and segment addresses and offsets must respect their alignment constraints (`sh_addralign`, `p_align`).

Memory Ownership and Cleanup

A clear ownership hierarchy prevents memory leaks:

- `elf_binary_t` **owns** the `buffer` (allocated via `malloc` / `realloc`)
- `elf_binary_t` **owns** the `sections` and `segments` arrays (allocated via `malloc`)
- Each `section_t` and `segment_t` **does not own** its `data` pointer (it's into `buffer`)
- `patch_site_t` **owns** its `new_bytes` and `nop_padding` (allocated via `malloc`)
- `trampoline_t` **owns** its `code` , `original_prologue` , and `relocated_prologue` (allocated via `malloc`)
- `injected_section_t` **owns** its `data` , `name` , and `symbol_dependencies` array (allocated via `malloc`)

Cleanup functions (`elf_binary_free()` , `patch_free()` , `trampoline_free()`) must follow this ownership hierarchy to free all allocated memory.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Data Structure Memory Management	Manual `malloc` / `free` with ownership documentation	Reference-counted smart pointers (C++/Rust) or arena allocator for batch operations
Error Propagation	Return error codes with `error_t` structure for details	Setjmp/longjmp for error unwinding or full exception handling (C++)
Collection Storage	Dynamic arrays with `realloc` for simplicity	Hash maps for symbol name lookup, balanced trees for patch offset management

Recommended File/Module Structure Place data structure definitions in a dedicated header to be shared across components:

```
elflex/
├── include/
│   ├── data_model.h     ← All structure definitions and typedefs
│   └── constants.h      ← ELF constants (SHT_*, SHF_*, PT_*, etc.)
├── src/
│   ├── data_model.c     ← Memory management functions (elf_binary_free, etc.)
│   ├── elf_loader.c     ← Creates elf_binary_t from file
│   ├── patch.c          ← patch_site_t operations
│   ├── trampoline.c     ← trampoline_t generation
│   └── inject.c         ← injected_section_t management
```

Infrastructure Starter Code (Complete) The following provides complete, ready-to-use definitions for the core ELF structures and error handling:

```c
/* constants.h - ELF constants for x86-64 */

#ifndef CONSTANTS_H

#define CONSTANTS_H


#include <stdint.h>


/* ELF magic */

#define EI_NIDENT 16

#define ELF_MAGIC "\x7f\x45\x4c\x46"


/* Section types */

#define SHT_NULL        0

#define SHT_PROGBITS    1

#define SHT_SYMTAB      2

#define SHT_STRTAB      3

#define SHT_NOBITS      8


/* Section flags */

#define SHF_WRITE       0x1

#define SHF_ALLOC       0x2

#define SHF_EXECINSTR   0x4


/* Program header types */

#define PT_LOAD         1


/* Error codes */

#define ERR_SUCCESS             0

#define ERR_INVALID_ELF         1

#define ERR_FILE_IO             2

#define ERR_OUT_OF_MEMORY       3

#define ERR_SECTION_NOT_FOUND   4

#define ERR_SYMBOL_NOT_FOUND    5

#define ERR_PATCH_OVERLAP       6

#define ERR_INVALID_OFFSET      7


#endif /* CONSTANTS_H */
```

```c
/* data_model.h - Core data structure definitions */

#ifndef DATA_MODEL_H

#define DATA_MODEL_H


#include <stddef.h>

#include <stdint.h>

#include "constants.h"


/* Standard ELF types (simplified for x86-64) */

typedef uint64_t Elf64_Addr;

typedef uint64_t Elf64_Off;

typedef uint16_t Elf64_Half;

typedef uint32_t Elf64_Word;

typedef uint64_t Elf64_Xword;


/* ELF header */

typedef struct {

    unsigned char e_ident[EI_NIDENT];

    Elf64_Half    e_type;

    Elf64_Half    e_machine;

    Elf64_Word    e_version;

    Elf64_Addr    e_entry;

    Elf64_Off     e_phoff;

    Elf64_Off     e_shoff;

    Elf64_Word    e_flags;

    Elf64_Half    e_ehsize;

    Elf64_Half    e_phentsize;

    Elf64_Half    e_phnum;

    Elf64_Half    e_shentsize;

    Elf64_Half    e_shnum;

    Elf64_Half    e_shstrndx;

} Elf64_Ehdr;


/* Section header */

typedef struct {

    Elf64_Word    sh_name;

    Elf64_Word    sh_type;
```

```
    Elf64_Xword   sh_flags;
    Elf64_Addr    sh_addr;
    Elf64_Off     sh_offset;
    Elf64_Xword   sh_size;
    Elf64_Word    sh_link;
    Elf64_Word    sh_info;
    Elf64_Xword   sh_addralign;
    Elf64_Xword   sh_entsize;
} Elf64_Shdr;

/* Program header */
typedef struct {
    Elf64_Word    p_type;
    Elf64_Word    p_flags;
    Elf64_Off     p_offset;
    Elf64_Addr    p_vaddr;
    Elf64_Addr    p_paddr;
    Elf64_Xword   p_filesz;
    Elf64_Xword   p_memsz;
    Elf64_Xword   p_align;
} Elf64_Phdr;

/* Symbol table entry */
typedef struct {
    Elf64_Word    st_name;
    unsigned char st_info;
    unsigned char st_other;
    Elf64_Half    st_shndx;
    Elf64_Addr    st_value;
    Elf64_Xword   st_size;
} Elf64_Sym;

/* Forward declarations */
typedef struct section_t section_t;
typedef struct segment_t segment_t;
typedef struct patch_site_t patch_site_t;
typedef struct trampoline_t trampoline_t;
```

```c
typedef struct injected_section_t injected_section_t;

typedef struct error_t error_t;


/* Section representation */

struct section_t {

    Elf64_Shdr*        shdr;

    char*              name;

    unsigned char*     data;

};


/* Segment representation */

struct segment_t {

    Elf64_Phdr*        phdr;

    unsigned char*     data;

};


/* ELF binary container */

typedef struct {

    unsigned char*     buffer;          /* Complete binary image */

    size_t             size;            /* Current buffer size */

    Elf64_Ehdr*        ehdr;            /* Pointer into buffer */

    section_t**        sections;        /* Array of section pointers */

    size_t             section_count;

    segment_t**        segments;        /* Array of segment pointers */

    size_t             segment_count;

    section_t*         text_section;    /* Cached .text section */

    section_t*         symtab;          /* Cached symbol table */

} elf_binary_t;


/* Patch metadata */

struct patch_site_t {

    uint64_t           target_offset;   /* File offset in buffer */

    size_t             original_size;   /* Size of region to replace */

    unsigned char*     new_bytes;       /* Replacement bytes */

    size_t             new_size;        /* Size of replacement */

    unsigned char*     nop_padding;     /* NOP sled for size mismatch */

    size_t             nop_padding_size;
```

```
    int                 applied;         /* 1 if applied to buffer */
};

/* Trampoline representation */
struct trampoline_t {
    unsigned char*      code;            /* Generated trampoline machine code */
    size_t              code_size;
    unsigned char*      original_prologue; /* Copy of displaced prologue */
    size_t              prologue_size;
    unsigned char*      relocated_prologue; /* Prologue with RIP fixups */
    size_t              relocated_size;
    uint64_t            hook_address;    /* Address of user hook function */
    uint64_t            target_function_address; /* Address of original function */
    uint64_t            trampoline_address; /* Where trampoline will be placed */
};

/* Injected section representation */
struct injected_section_t {
    char*               name;            /* Section name (e.g., ".inject") */
    unsigned char*      data;            /* Raw section content */
    size_t              data_size;
    uint64_t            vaddr;           /* Assigned virtual address */
    Elf64_Shdr*         shdr;            /* Pointer to header in buffer */
    int                 requires_relocation; /* Non-zero if relocations needed */
    char**              symbol_dependencies; /* Symbols needing resolution */
    size_t              dependency_count;
};

/* Error structure */
struct error_t {
    int                 code;            /* ERR_* constant */
    char                message[256];    /* Human-readable description */
    char                context[128];    /* Where error occurred */
};

#endif /* DATA_MODEL_H */
```

Core Logic Skeleton Code The following skeleton shows the memory management functions with TODOs that learners must implement:

C

```c
/* data_model.c - Memory management for data structures */

#include <stdlib.h>

#include <string.h>

#include "data_model.h"


/* Create a new empty error structure */

error_t* error_create() {

    error_t* err = malloc(sizeof(error_t));

    if (!err) return NULL;

    err->code = ERR_SUCCESS;

    err->message[0] = '\0';

    err->context[0] = '\0';

    return err;

}


/* Free an error structure */

void error_free(error_t* err) {

    free(err);

}


/* Free an ELF binary and all its owned resources */

void elf_binary_free(elf_binary_t* binary) {

    if (!binary) return;



    // TODO 1: Free the sections array (but not section_t objects themselves,

    //          as they are part of the buffer)

    // Hint: binary->sections is an array of pointers, but each section_t

    //       is located within the buffer, not separately allocated



    // TODO 2: Free the segments array (similar to sections)



    // TODO 3: Free the main buffer



    // TODO 4: Free the binary structure itself

}


/* Free a patch site and its owned resources */
```

```c
void patch_free(patch_site_t* patch) {
    if (!patch) return;

    // TODO 1: Free new_bytes if allocated

    // TODO 2: Free nop_padding if allocated

    // TODO 3: Free the patch structure itself
}

/* Free a trampoline and its owned resources */
void trampoline_free(trampoline_t* trampoline) {
    if (!trampoline) return;

    // TODO 1: Free code buffer

    // TODO 2: Free original_prologue buffer

    // TODO 3: Free relocated_prologue buffer

    // TODO 4: Free the trampoline structure itself
}

/* Free an injected section and its owned resources */
void injected_section_free(injected_section_t* section) {
    if (!section) return;

    // TODO 1: Free the section name

    // TODO 2: Free the data buffer

    // TODO 3: Free symbol_dependencies array (each string and the array itself)

    // TODO 4: Free the section structure itself
}
```

Language-Specific Hints (C)

- **Structure Packing:** ELF structures are tightly packed with 1-byte alignment. Use `#pragma pack(1)` or compiler attributes if your compiler adds padding.
- **Pointer Arithmetic:** When calculating pointers into the `buffer`, use `(unsigned char*)` arithmetic: `Elf64_Ehdr* ehdr = (Elf64_Ehdr*)(buffer + 0)`.
- **Endianness:** The data model assumes host endianness matches the target ELF (little-endian for x86-64). For cross-architecture support, you'd need byte-swapping functions.
- **Dynamic Arrays:** Use `realloc` for growing arrays like `sections` and `segments` when adding new sections during injection.

Milestone Checkpoint (Data Model) After implementing the data structures and memory management functions:

- **Test Command:** `cc -c src/data_model.c -Iinclude && echo "Data model compiles successfully"`
- **Expected Output:** No compilation errors or warnings.
- **Verification:** Write a simple test program that allocates and frees each structure type, then run under Valgrind: `valgrind ./test_data_model`
- **Success Indicator:** Valgrind reports "0 bytes in 0 blocks" lost (no memory leaks).
- **Common Failure:** Segmentation fault when freeing - indicates incorrect ownership model or double-free. Check that pointers are either `NULL` or valid before freeing.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
`buffer` corruption after multiple patches	Patch offsets calculated incorrectly after buffer reallocation	Print `target_offset` before and after each patch, check if it's still within `buffer` size	Recalculate offsets after buffer reallocation using section header information
Memory leak when injecting sections	`symbol_dependencies` array not fully freed	Use Valgrind with `--leak-check=full` to see which allocation isn't freed	Ensure `injected_section_free` frees each string in the array before freeing the array itself
Invalid pointer read in `section_t->data`	Section data pointer not updated after `buffer` reallocation	Compare `section->data` with `binary->buffer + section->shdr->sh_offset`	After `realloc` of buffer, recalculate all `data` pointers using `sh_offset` / `p_offset`
Trampoline code crashes when executed	`relocated_prologue` contains incorrect RIP-relative fixups	Disassemble the relocated prologue with `objdump -D -b binary` and compare with original	Implement `fixup_rip_relative` to properly adjust displacements

---

Component: ELF Loader and Parser

> **Milestone(s):** 1 (Instruction-level Patching)

The **ELF Loader and Parser** is the foundational component that transforms a binary ELF file from an opaque byte stream into a structured, navigable, and modifiable in-memory representation. Every modification ELFlex makes—whether patching an instruction, hooking a function, or injecting new code—begins with this component's ability to accurately parse and understand the binary's internal structure.

Mental Model: The Map Reader

Think of a compiled ELF binary as a **detailed architectural blueprint for a complex building**. The blueprint contains multiple overlays: foundation plans (headers), floor layouts (segments), room specifications (sections), and labeled access points (symbols). You cannot modify the building by randomly drilling holes; you need to understand where the load-bearing walls (code) are, where electrical wiring (data) runs, and which doors (function entry points) you can intercept.

The ELF Loader is the **cartographer** who translates this blueprint into an **editable, annotated map**. It doesn't just read bytes; it constructs a mental model of the binary's layout, identifying:

- Where executable code resides (`.text` section)
- Where global variables live (`.data` , `.bss`)
- Where function names and their addresses are listed (`.symtab`)
- How the operating system will load the binary into memory (program headers)
- Where each piece of content begins and ends in the file (section headers)

This map enables the subsequent components to perform precise, surgical modifications without accidentally damaging the binary's structural integrity.

Interface and Responsibilities

The ELF Loader component exposes a clean API that hides the complexity of ELF parsing while providing essential query and navigation capabilities to other components.

Method Name	Parameters	Returns	Description
`elf_load`	`filename char*`	`elf_binary_t*`	Loads an ELF file from disk, validates its structure, and builds the complete in-memory representation. Returns `NULL` on failure.
`elf_find_section`	`binary elf_binary_t*` , `name char*`	`section_t*`	Searches for a section by its name (e.g., ".text", ".symtab") in the section header table. Returns `NULL` if not found.
`elf_get_symbol_address`	`binary elf_binary_t*` , `symbol_name char*`	`uint64_t`	Looks up a symbol (function or variable name) in the symbol table and returns its virtual address. Returns `0` if symbol not found or address is zero.
`elf_binary_free`	`binary elf_binary_t*`	`void`	Releases all memory associated with the binary representation, including sections, segments, and the underlying buffer.
`is_valid_elf`	`buffer unsigned char*` , `size size_t`	`int`	Validates that the buffer contains a valid ELF file by checking the magic number, header sizes, and basic consistency. Returns non-zero if valid.
`elf_error`	`message char*`	`void`	Reports an ELF parsing error, typically by printing to stderr with contextual information. Used internally for debugging.

The primary output of this component is an `elf_binary_t` structure, which serves as the root of the entire data model for subsequent modifications. This structure maintains ownership of the original file buffer and all parsed metadata.

Internal Behavior and Algorithms

When `elf_load` is called, it executes a multi-stage parsing pipeline that transforms raw bytes into a structured representation:

1. **File Reading and Initial Validation**
   - The entire file is read into a contiguous memory buffer via `read_file` .
   - `is_valid_elf` checks the first 4 bytes for the ELF magic number (`\x7fELF`).
   - The ELF header (`Elf64_Ehdr`) is validated: checking the class (64-bit), data encoding (little/big endian), version, and that it's an executable or shared object.
2. **Program Header Table Parsing**
   - Using `e_phoff` (program header table offset) and `e_phnum` (number of entries), each `Elf64_Phdr` is read.
   - For each `PT_LOAD` segment (the segments actually loaded into memory), a `segment_t` is created containing the header and a pointer to the segment's data within the file buffer.
   - Segment data is **not** copied by default; the `data` field points into the original buffer. This avoids unnecessary duplication for read-only analysis.
3. **Section Header Table Parsing**

- Using `e_shoff` (section header table offset) and `e_shnum`, each `Elf64_Shdr` is read.
- The section name string table (index `e_shstrndx`) is located to resolve `sh_name` fields into human-readable names.
- For each section, a `section_t` is created with its header, name, and data pointer (again, pointing into the original buffer).
- Special attention is given to locating the `.text` section (executable code) and `.symtab` (symbol table), which are cached in the `elf_binary_t` for fast access.

4. **Symbol Table Processing**
   - When the `.symtab` section is found, its entries (each `Elf64_Sym`) are parsed but not fully loaded into a separate structure initially. Instead, the component stores a pointer to the symbol table data and its associated string table (`.strtab`) for on-demand lookup.
   - The symbol table's entries are validated to ensure they reference valid sections and have reasonable addresses.
5. **Consistency Cross-Checks**
   - The loader verifies that section addresses and sizes don't contradict segment mappings.
   - It ensures that the `.text` section falls within a `PT_LOAD` segment with execute permissions (`PF_X`).
   - Virtual address ranges are checked for overlaps (though legitimate overlaps can occur with different permissions).

> **Key Insight:** The parser adopts a **lazy evaluation** strategy for symbol resolution and relocation processing. Full symbol table parsing only occurs when `elf_get_symbol_address` is called, not during initial load. This keeps the common case (loading for patching) fast and memory-efficient.

ADR: In-Memory vs. On-Disk Representation

Decision: Maintain a complete in-memory copy of the binary with editable buffers

Option	Pros	Cons
In-memory copy with separate editable buffers	• Easy to modify any part of the binary without complex offset tracking • Can preview changes before writing to disk • Simplifies error recovery (discard buffer if patch fails)	• Higher memory usage (entire file plus modification buffers) • Requires careful synchronization between different representations
Streaming edits to disk	• Minimal memory footprint • No need to hold entire binary in RAM	• Extremely complex offset management when inserting/deleting bytes • Error recovery is difficult (partial writes corrupt the file) • Cannot easily preview or validate changes before committing
Hybrid: copy only modified sections	• Balances memory usage and simplicity • Only affected sections need duplication	• Still requires tracking offsets across the file • Complex when modifications affect multiple sections or headers

Context: ELFlex needs to support multiple modification types—from simple instruction replacement (Milestone 1) to adding entire new sections (Milestone 3). These operations require changing various parts of the ELF file simultaneously: code, section headers, program headers, and possibly symbol tables. The complexity of maintaining consistency while streaming edits to disk would be substantial.

Decision: We maintain the entire original file in a memory buffer and create separate, editable copies of sections that will be modified. The `elf_binary_t` structure points to the original buffer for unmodified sections and to new allocations for modified ones.

Rationale:

1. **Simpler offset management:** When we add a new section or expand `.text`, we can compute all new offsets once and apply them to our in-memory structures before writing anything to disk.
2. **Atomic operations:** Either all modifications succeed (and we write a complete new file) or none do (we discard the buffers). This prevents partially-patched, corrupted binaries.
3. **Debugging visibility:** Having the complete modified binary in memory allows us to add validation and debugging outputs before committing to disk.

Consequences:

- Memory usage scales with binary size (a 10MB binary requires ~10MB plus modification overhead).
- The `binary_write` function must reconstruct the entire ELF file from the in-memory representation, which involves recomputing offsets and sizes for all sections and segments.
- We must implement careful memory ownership rules to avoid leaks when discarding failed modifications.

Common Pitfalls: Alignment and Endianness

⚠️ **Pitfall: Ignoring Section Alignment Requirements** ELF sections have `sh_addralign` fields that specify alignment constraints (often 16 bytes for `.text`). When modifying section sizes or adding new sections, failing to maintain proper alignment can cause the loader to reject the binary or cause runtime crashes due to misaligned memory accesses.

Why it's wrong: The operating system's program loader uses these alignment values to position sections in memory. Misalignment can lead to inefficient memory usage (wasted space) or, on some architectures, alignment faults when accessing data.

How to fix: Always round up section sizes to the next multiple of their alignment. When adding new sections, choose an alignment that matches the target architecture's page size (typically 4096) or the largest alignment in existing sections.

⚠️ **Pitfall: Assuming Host Endianness** ELF files can be little-endian (common on x86) or big-endian (some ARM, MIPS, PowerPC). Parsing header fields without considering endianness will work on matching architectures but fail catastrophically when the host and target differ.

Why it's wrong: All multi-byte values in ELF headers (except the magic number) are stored in the target architecture's endianness. Reading a 64-bit address as raw bytes on a mismatched system produces garbage values.

How to fix: Check the `e_ident[EI_DATA]` field in the ELF header (1 = little, 2 = big). Use conversion functions like `le64toh` or `be64toh` (from `endian.h`) when reading values from the buffer. Better yet, define your ELF structs with the correct endianness annotations or write a generic parser that handles both.

⚠️ **Pitfall: Misparsing the Symbol Table** The symbol table contains variable-length entries where the `st_name` field is an offset into the string table, not a pointer. Additionally, some symbols (like `STT_SECTION` or `STT_FILE`) don't represent functions and should be ignored for hooking purposes.

Why it's wrong: Treating `st_name` as a direct pointer will read random memory. Including non-function symbols in hooking targets will attempt to patch invalid addresses or data sections.

How to fix: Always resolve symbol names via `sh_link` to find the associated string table. When looking for functions to hook, check `st_info` for `STT_FUNC` type and `st_shndx` for a valid section index (not `SHN_UNDEF`).

⚠️ **Pitfall: Overlooking Relocation Sections** Many ELF binaries contain `.rela.text` or similar relocation sections that specify address adjustments needed at load time. These relocations affect absolute addresses in the code, which may need updating if we move code around.

Why it's wrong: If we patch or relocate code that contains relocated addresses, those addresses may become incorrect unless we also update the relocation entries.

How to fix: During initial parsing, identify and parse relocation sections. When modifying code, check if any relocation entries target the modified region and adjust them accordingly. For Milestone 1 (simple patching), we can require that patches don't affect relocated instructions.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
ELF Parsing	Manual struct mapping with `memcpy`	libelf (full-featured library)
Endianness Handling	`endian.h` conversion functions	Compiler intrinsics with `__builtin_bswap`
Memory Management	Manual `malloc` / `free` with clear ownership	Reference-counted smart pointers (not in C)

Recommended File/Module Structure:

```
elflex/
├── src/
│   ├── main.c                    # Entry point and CLI
│   ├── elf/                      # ELF Loader component
│   │   ├── elf_loader.c          # Main loading logic
│   │   ├── elf_loader.h          # Public interface
│   │   ├── elf_parser.c          # Header parsing internals
│   │   └── elf_parser.h          # Internal structs and helpers
│   ├── patch/                    # Patch Planner component
│   ├── trampoline/               # Trampoline Generator
│   └── utils/
│       ├── file_io.c             # read_file, write_file
│       └── error.c               # error_t utilities
└── tests/
    └── test_elf_loader.c         # Unit tests for this component
```

Infrastructure Starter Code:

```c
/* src/utils/file_io.c */

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "utils/file_io.h"


unsigned char* read_file(const char* filename, size_t* out_size) {

    FILE* file = fopen(filename, "rb");

    if (!file) return NULL;



    fseek(file, 0, SEEK_END);

    long file_size = ftell(file);

    fseek(file, 0, SEEK_SET);



    if (file_size <= 0) {

        fclose(file);

        return NULL;

    }



    unsigned char* buffer = malloc(file_size);

    if (!buffer) {

        fclose(file);

        return NULL;

    }



    size_t read = fread(buffer, 1, file_size, file);

    fclose(file);



    if (read != (size_t)file_size) {

        free(buffer);

        return NULL;

    }



    *out_size = (size_t)file_size;

    return buffer;

}
```

```c
int write_file(const char* filename, unsigned char* buffer, size_t size) {
    FILE* file = fopen(filename, "wb");
    if (!file) return 0;

    size_t written = fwrite(buffer, 1, size, file);
    fclose(file);

    return written == size;
}
```

```c
/* src/elf/elf_parser.h */

#ifndef ELF_PARSER_H

#define ELF_PARSER_H


#include <stdint.h>


#define EI_NIDENT 16


/* Standard ELF types - match system elf.h if available */

typedef uint64_t Elf64_Addr;

typedef uint64_t Elf64_Off;

typedef uint16_t Elf64_Half;

typedef uint32_t Elf64_Word;

typedef int32_t  Elf64_Sword;

typedef uint64_t Elf64_Xword;

typedef int64_t  Elf64_Sxword;


/* ELF Header */

typedef struct {

    unsigned char e_ident[EI_NIDENT];

    Elf64_Half    e_type;

    Elf64_Half    e_machine;

    Elf64_Word    e_version;

    Elf64_Addr    e_entry;

    Elf64_Off     e_phoff;

    Elf64_Off     e_shoff;

    Elf64_Word    e_flags;

    Elf64_Half    e_ehsize;

    Elf64_Half    e_phentsize;

    Elf64_Half    e_phnum;

    Elf64_Half    e_shentsize;

    Elf64_Half    e_shnum;

    Elf64_Half    e_shstrndx;

} Elf64_Ehdr;


/* Section Header */

typedef struct {

    Elf64_Word   sh_name;
```

```
    Elf64_Word   sh_type;
    Elf64_Xword  sh_flags;
    Elf64_Addr   sh_addr;
    Elf64_Off    sh_offset;
    Elf64_Xword  sh_size;
    Elf64_Word   sh_link;
    Elf64_Word   sh_info;
    Elf64_Xword  sh_addralign;
    Elf64_Xword  sh_entsize;
} Elf64_Shdr;

/* Program Header */
typedef struct {
    Elf64_Word   p_type;
    Elf64_Word   p_flags;
    Elf64_Off    p_offset;
    Elf64_Addr   p_vaddr;
    Elf64_Addr   p_paddr;
    Elf64_Xword  p_filesz;
    Elf64_Xword  p_memsz;
    Elf64_Xword  p_align;
} Elf64_Phdr;

/* Symbol Table Entry */
typedef struct {
    Elf64_Word    st_name;
    unsigned char st_info;
    unsigned char st_other;
    Elf64_Half    st_shndx;
    Elf64_Addr    st_value;
    Elf64_Xword   st_size;
} Elf64_Sym;

/* Constants */
#define ELF_MAGIC "\x7f\x45\x4c\x46"
#define SHT_NULL      0
#define SHT_PROGBITS  1
```

```
#define SHT_SYMTAB    2

#define SHT_STRTAB    3

#define SHT_NOBITS    8

#define SHF_WRITE     0x1

#define SHF_ALLOC     0x2

#define SHF_EXECINSTR 0x4

#define PT_LOAD       1


#endif /* ELF_PARSER_H */
```

Core Logic Skeleton Code:

```c
/* src/elf/elf_loader.c */

#include "elf/elf_loader.h"

#include "elf/elf_parser.h"

#include "utils/file_io.h"

#include <stdlib.h>

#include <string.h>


/* Internal helper to convert endianness if needed */

static uint64_t elf64_to_host(uint64_t value, int is_little_endian) {

    /* TODO 1: Implement endianness conversion

       If host is little-endian and binary is little-endian, return value as-is

       If endianness differs, byte-swap the value

       Hint: Use __builtin_bswap64 or manual byte swapping */

    return value;

}


int is_valid_elf(unsigned char* buffer, size_t size) {

    /* TODO 2: Validate ELF file

       Check if size >= sizeof(Elf64_Ehdr)

       Check magic bytes at buffer[0..3] == ELF_MAGIC

       Check ELF class (64-bit) at buffer[EI_CLASS] == 2

       Return 1 if valid, 0 otherwise */

    return 0;

}


elf_binary_t* elf_load(const char* filename) {

    /* TODO 3: Read file into buffer */

    size_t file_size;

    unsigned char* buffer = read_file(filename, &file_size);

    if (!buffer) return NULL;



    /* TODO 4: Validate it's a valid ELF */

    if (!is_valid_elf(buffer, file_size)) {

        free(buffer);

        return NULL;

    }
```

```c
/* TODO 5: Allocate and initialize elf_binary_t */
elf_binary_t* binary = calloc(1, sizeof(elf_binary_t));
if (!binary) {
    free(buffer);
    return NULL;
}
binary->buffer = buffer;
binary->size = file_size;

/* TODO 6: Parse ELF header */
Elf64_Ehdr* ehdr = (Elf64_Ehdr*)buffer;
binary->ehdr = ehdr;

/* Determine endianness for future conversions */
int is_little = (ehdr->e_ident[EI_DATA] == 1); /* 1 = ELFDATA2LSB */

/* TODO 7: Parse program headers (if any) */
if (ehdr->e_phnum > 0) {
    binary->segments = malloc(sizeof(segment_t*) * ehdr->e_phnum);
    /* TODO 7a: For each program header, create segment_t */
    /* TODO 7b: Store pointer to segment data in buffer */
    /* TODO 7c: Count actual PT_LOAD segments for segment_count */
}

/* TODO 8: Parse section headers (if any) */
if (ehdr->e_shnum > 0) {
    binary->sections = malloc(sizeof(section_t*) * ehdr->e_shnum);
    /* TODO 8a: Find string table for section names (e_shstrndx) */
    /* TODO 8b: For each section header, create section_t */
    /* TODO 8c: Resolve section name using string table */
    /* TODO 8d: Store pointer to section data in buffer */
    /* TODO 8e: Special handling for .text and .symtab sections */
}

return binary;
```

```
}

section_t* elf_find_section(elf_binary_t* binary, const char* name) {
    /* TODO 9: Find section by name
       Iterate through binary->sections
       Compare section_t->name with target name
       Return first match or NULL */
    return NULL;
}

uint64_t elf_get_symbol_address(elf_binary_t* binary, const char* symbol_name) {
    /* TODO 10: Locate .symtab and associated .strtab sections */
    /* TODO 11: Iterate through symbol table entries (Elf64_Sym) */
    /* TODO 12: For each symbol, resolve name via string table offset */
    /* TODO 13: Compare with symbol_name, return st_value if match */
    /* TODO 14: Only return symbols of type STT_FUNC (functions) */
    return 0;
}

void elf_binary_free(elf_binary_t* binary) {
    if (!binary) return;

    /* TODO 15: Free all allocated resources
       Free each section_t (but NOT the data pointers into buffer)
       Free each segment_t
       Free sections and segments arrays
       Free the original buffer
       Free binary itself */
}
```

Language-Specific Hints:

1. **Memory Ownership:** The `elf_binary_t` owns the `buffer` (allocated by `read_file`). Sections and segments point into this buffer. When modifying a section, allocate new memory and update the section's `data` pointer to point to the new allocation.
2. **Endianness in C:** Use `#include <endian.h>` for `le64toh` and `be64toh` functions. Alternatively, detect host endianness at compile time and implement your own swapping functions.
3. **Pointer Arithmetic:** When calculating offsets into the buffer, cast to `uintptr_t` before arithmetic: `(unsigned char*)((uintptr_t)buffer + offset)` to avoid potential alignment issues.
4. **String Table Handling:** String table entries are null-terminated strings. The `sh_name` and `st_name` fields are offsets from the beginning of the string table. Use `strcmp` for comparison once you have the string pointer.

Milestone Checkpoint:

After implementing the ELF Loader, you should be able to:

1. Load a simple test binary: `./elflex load test_binary`
2. Verify it correctly identifies the `.text` section: Expect output showing section virtual address and size.
3. Look up a known function symbol: `./elflex symbol test_binary main` should print the address of `main`.

Verification Commands:

- Compare with `readelf -S test_binary` to check section parsing.
- Compare with `readelf -s test_binary | grep main` to verify symbol lookup.

Signs of Trouble:

- **Segmentation fault when loading:** Likely incorrect pointer arithmetic or misaligned struct access. Use `gdb` to trace which line crashes.
- **Wrong section addresses:** Endianness issue. Check `EI_DATA` field and implement conversion.
- **Missing .text section:** The section name string table index (`e_shstrndx`) might be wrong or the section might have a different name (some compilers use `.text.startup`).

Component: Patch Planner and Binary Patcher

> **Milestone(s):** 1 (Instruction-level Patching)

The **Patch Planner and Binary Patcher** is the surgical component responsible for making precise, low-level modifications to executable code within ELF binaries. While the ELF Loader provides the map and understanding of the binary's structure, this component wields the scalpel that actually changes the machine instructions. Its primary challenge is modifying compiled code without breaking the binary's structural integrity or control flow logic, requiring careful planning, size management, and awareness of architectural constraints.

Mental Model: The Surgical Editor

Imagine you're editing a printed book to add a footnote reference within a paragraph. You can't change the page numbers or paragraph lengths without affecting the entire book's layout. The **Patch Planner and Binary Patcher** operates like a surgical text editor for machine code:

1. **Text as Code**: The `.text` section is the "paragraph" containing the narrative (program logic). Each instruction is a "word" with fixed meaning and precise placement.
2. **Size-Preserving Edits**: Just as you can't insert words without changing line breaks, you can't insert instructions without affecting subsequent instruction addresses. The patcher maintains the original section size by replacing instructions with equivalent-sized alternatives or padding with "filler words" (NOP instructions).
3. **Grammar Preservation**: Machine code has strict "grammar" rules—branch targets must remain valid, relative offsets must point to the right places, and instruction boundaries must be respected. The patcher ensures these rules aren't violated.
4. **Planning Before Cutting**: Before making any changes, the patcher creates a surgical plan (`patch_site_t`) that specifies exactly what to replace, what to insert, and how to handle size mismatches. This prevents accidental damage to adjacent code.

This mental model emphasizes the patcher's dual role: it must be both **precise** (changing only what's needed) and **non-destructive** (maintaining everything else exactly as was).

Interface and Responsibilities

The Patch Planner and Binary Patcher exposes a focused API for planning and applying instruction-level modifications. Its responsibilities center around validating, planning, and executing binary patches while maintaining ELF structural correctness.

Method Name	Parameters	Returns	Description
`plan_patch`	`binary` : `elf_binary_t*` `offset` : `uint64_t` `new_bytes` : `unsigned char*` `new_size` : `size_t`	`patch_site_t*`	Analyzes a requested patch at the given file offset within the binary's `.text` section. Validates that the offset is within executable code, compares instruction sizes, and generates a patch plan including any necessary NOP padding. Returns a complete `patch_site_t` structure ready for application.
`apply_patch`	`binary` : `elf_binary_t*` `patch` : `patch_site_t*`	`int` (error code)	Executes a planned patch by modifying the in-memory representation of the binary. Copies new bytes (and optional NOP padding) into the target location, updates the patch's `applied` flag, and ensures the binary remains internally consistent. Returns `ERR_SUCCESS` on success or an appropriate error code.
`generate_nop_sled`	`count` : `size_t`	`unsigned char*`	Creates a sequence of NOP (No Operation) instructions of the specified length. The generated sled is architecture-specific (x86-64 NOP is `0x90`). The caller must free the returned buffer.
`patch_free`	`patch` : `patch_site_t*`	`void`	Deallocates all memory associated with a patch site, including the `new_bytes` and `nop_padding` buffers if they were allocated by `plan_patch`.

The component also interacts indirectly with the ELF Loader to query section boundaries and with the Binary Writer to persist changes to disk. Its design follows the principle of **separation of concerns**: planning is pure computation, while application mutates state.

Internal Behavior: The Patching Algorithm

The patching process follows a deterministic sequence that ensures safety and correctness. Below is the step-by-step algorithm executed by `plan_patch` and `apply_patch`:

Phase 1: Planning (`plan_patch`)

1. **Offset Validation**: Verify the target `offset` falls within a loadable, executable segment (typically `.text`). Check that `offset + new_size` doesn't exceed the segment boundary. If invalid, return `ERR_INVALID_OFFSET`.
2. **Instruction Boundary Check**: Ensure the offset is at a valid instruction boundary (multiple of instruction size alignment, typically 1 byte for x86). While x86 allows unaligned instructions, patching in the middle of a multi-byte instruction would corrupt execution.
3. **Size Analysis**: Compare `new_size` with the space available from `offset` to the end of the containing basic block or function (estimated via simple disassembly or conservative fixed chunk). For Milestone 1's simple case, we assume we're replacing a contiguous block of original instructions whose total size we know.
4. **Padding Strategy**:
   - If `new_size` equals the original instruction block size: no padding needed.
   - If `new_size` is smaller: generate a NOP sled of size `(original_size - new_size)` using `generate_nop_sled`.
   - If `new_size` is larger: this is an error for the initial implementation (see ADR below). Return `ERR_PATCH_OVERLAP`.
5. **Patch Plan Construction**: Create and populate a `patch_site_t` structure:
   - `target_offset` : The file offset to patch
   - `original_size` : Size of instructions being replaced (determined by caller or via disassembly)
   - `new_bytes` : Copy of the provided byte sequence
   - `new_size` : As provided
   - `nop_padding` : Generated NOP sled if needed
   - `nop_padding_size` : Size of NOP sled
   - `applied` : Set to 0 (not yet applied)

6. **Overlap Detection**: Check that this patch doesn't overlap with any previously applied patches (requires maintaining patch list in binary or having caller manage). Overlapping patches would create ambiguous execution behavior.

Phase 2: Application (`apply_patch`)

1. **Precondition Check**: Verify the patch hasn't already been applied (`applied == 0`) and that the binary's in-memory buffer is writable.
2. **Memory Safety**: Ensure the target offset range `[target_offset, target_offset + original_size)` is within the binary's buffer bounds.
3. **Byte Replacement**:
   - Copy `new_bytes` to `binary->buffer + target_offset`
   - If `nop_padding_size > 0` , copy `nop_padding` to `binary->buffer + target_offset + new_size`
4. **State Update**: Set `patch->applied = 1` to prevent reapplication.
5. **Integrity Preservation**: The binary's section and segment headers remain unchanged because we're modifying content within existing sections without changing their sizes or offsets.

Concrete Walk-Through Example

Consider patching a simple function that begins at file offset `0x400` with the following original instructions (5 bytes total):

```
55                  push rbp
48 89 e5            mov rbp, rsp
b8 2a 00 00 00      mov eax, 0x2a
```

We want to replace the `mov eax, 0x2a` (5 bytes) with `xor eax, eax` (2 bytes, bytes `0x31 0xc0`). The planning phase:

1. Offset `0x40a` (start of third instruction) is validated as within `.text`
2. Original instruction size is 5 bytes, new size is 2 bytes
3. Size mismatch: generate NOP sled of 3 bytes (`0x90 0x90 0x90`)
4. Construct patch plan with `target_offset=0x40a` , `original_size=5` , `new_size=2` , `nop_padding_size=3`

Application phase writes `0x31 0xc0 0x90 0x90 0x90` at offset `0x40a` . The function now executes `xor eax, eax` (sets eax to 0) followed by three NOPs, maintaining total code size.

ADR: Same-Size vs. Variable-Size Patching

> **Decision: Enforce Same-Size Replacements with NOP Padding for Initial Implementation**
>
> - **Context**: When replacing instructions in a compiled binary, the new instruction sequence may be smaller or larger than the original. Larger replacements require shifting subsequent code, which changes instruction addresses and breaks relative jumps. Smaller replacements leave gaps that could be filled with NOPs or require compaction.
> - **Options Considered**:
>   1. **Strict same-size replacement**: Require caller to provide new bytes exactly matching original size.
>   2. **NOP-padded same-size**: Allow size mismatch but pad with NOPs to maintain total size.
>   3. **Variable-size with code relocation**: Allow any size and shift subsequent code, updating all affected addresses and relocations.
> - **Decision**: Option 2 (NOP-padded same-size) for Milestone 1, with Option 3 as a future extension.
> - **Rationale**:
>   - **Simplicity**: NOP padding requires only local changes at the patch site, avoiding complex global address recalculations.
>   - **Safety**: Preserves all existing relative offsets (jumps, calls, RIP-relative addressing) since no instructions move.
>   - **Gradual Complexity**: Provides a working foundation for instruction patching before tackling the more complex problem of code relocation required for variable-size patches.
>   - **Practicality**: Many instrumentation use cases (e.g., inserting jump instructions for hooks) involve replacing a fixed-size prologue with another fixed-size sequence plus NOPs.
> - **Consequences**:
>   - **Positive**: Implementation is simpler, faster, and less error-prone. No need for disassembly to identify all affected branches.
>   - **Negative**: Inefficient use of space (NOP sleds waste bytes). Cannot insert larger code sequences without later compaction.
>   - **Forward Compatibility**: The design leaves room to implement Option 3 later by extending the patch planner to track and adjust relative offsets.

Option	Pros	Cons	Chosen?
Strict same-size replacement	Simplest implementation, no wasted space	Inflexible, requires exact size matching by caller	No
NOP-padded same-size	**Flexible for common cases, preserves all offsets, simpler than full relocation**	**Wastes space with NOPs, cannot expand beyond original size**	**Yes (Milestone 1)**
Variable-size with relocation	Most flexible, efficient space usage	Complex: requires disassembly, offset updates, potential for errors	No (future)

This decision aligns with the project's incremental approach: start with a working, safe foundation before adding complexity.

Common Pitfalls

⚠️ Pitfall: Patching Across Instruction Boundaries

Description: Applying a patch that starts or ends in the middle of a multi-byte instruction. For example, patching 2 bytes starting at offset that is the second byte of a 3-byte instruction.

Why It's Wrong: x86 instructions have variable length (1-15 bytes). Patching partial instructions creates invalid opcodes or unintended instruction sequences that will crash or behave unpredictably when executed.

How to Avoid: Always patch at known instruction boundaries. Use a disassembler (even a simple one) to identify instruction boundaries, or rely on symbol/section information that typically aligns functions to instruction boundaries. In `plan_patch`, validate that the offset is at a function entry point (from symbol table) or use conservative heuristics.

⚠️ Pitfall: Ignoring Relative Jump Offsets

Description: Patching code that contains relative jumps (`jmp`, `call`, conditional jumps) without adjusting their target offsets if the jump instruction itself moves.

Why It's Wrong: Relative jumps encode the distance to the target as an offset from the next instruction's address. If you insert or remove bytes between the jump and its target (by changing sizes), the offset becomes incorrect, causing jumps to wrong locations.

How to Avoid: In the NOP-padded approach, jumps don't move relative to their targets, so this isn't an issue. However, if patching the jump instruction itself (e.g., replacing a `jmp` with a different one), ensure the new jump has the correct offset. For future variable-size patching, you'll need to disassemble to identify all relative jumps and recalculate their offsets.

⚠️ Pitfall: Instruction Cache Incoherence

Description: Modifying code in memory (either in the binary buffer or at runtime) without ensuring the CPU's instruction cache is synchronized with the new code.

Why It's Wrong: Modern CPUs cache decoded instructions. If you modify code in memory but the cache still holds old versions, the CPU may execute stale instructions, leading to undefined behavior or crashes.

How to Avoid:

- For on-disk patching: The OS will load the fresh code from disk when the binary is executed, so cache isn't an issue.
- For runtime patching (Milestone 4): Use `__builtin___clear_cache()` on Linux or architecture-specific cache flush instructions after writing to code pages.

⚠️ Pitfall: Overlapping Patches

Description: Applying a second patch that overlaps with a previously applied patch's byte range.

Why It's Wrong: Creates ambiguous execution behavior—which patch's bytes take effect? The result depends on application order and may corrupt intended modifications.

How to Avoid: Maintain a registry of applied patches in the `elf_binary_t` structure or require the caller to manage patch non-overlap. In `plan_patch`, check if the requested offset range intersects with any existing patch's range.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Instruction Patching	Byte-by-byte memory copy with NOP padding	Use Capstone/ Zydis disassembler for instruction boundary validation
NOP Generation	Hardcoded x86-64 NOP (`0x90`)	Architecture-aware NOP sequences for different ISAs (ARM, RISC-V)
Patch Management	Linear array of patches in `elf_binary_t`	Interval tree for efficient overlap detection

B. Recommended File/Module Structure

```
elflex/
├── src/
│   ├── main.c                    # CLI entry point
│   ├── elf/                      # ELF parsing (previous component)
│   │   ├── loader.c
│   │   └── loader.h
│   ├── patch/                    # This component
│   │   ├── planner.c
│   │   ├── planner.h
│   │   └── nop.c                 # NOP generation utilities
│   ├── trampoline/               # Milestone 2 component
│   └── runtime/                  # Milestone 4 component
└── tests/
    ├── test_patch.c
    └── test_binaries/            # Simple test binaries for patching
```

C. Infrastructure Starter Code

File: `src/patch/nop.c` (Complete working code)

```c
#include <stdlib.h>

#include <string.h>

#include "patch/planner.h"


/**

 * Generate a sequence of NOP instructions for x86-64 architecture.

 *

 * @param count Number of NOP bytes to generate

 * @return Dynamically allocated buffer containing NOP instructions

 * @note Caller must free the returned buffer with free()

 */

unsigned char* generate_nop_sled(size_t count) {

    if (count == 0) {

        return NULL;

    }



    unsigned char* nop_buffer = (unsigned char*)malloc(count);

    if (nop_buffer == NULL) {

        return NULL;

    }



    // x86-64 NOP instruction is 0x90

    // For better performance on some CPUs, we could use multi-byte NOPs

    // like 0x66 0x90 (2-byte), 0x0f 0x1f 0x00 (3-byte), etc.

    // But for simplicity, we use single-byte NOPs.

    memset(nop_buffer, 0x90, count);



    return nop_buffer;

}
```

D. Core Logic Skeleton Code

File: `src/patch/planner.c` (Skeleton with TODOs)

```c
#include <stdlib.h>

#include <string.h>

#include <stdint.h>

#include "patch/planner.h"

#include "elf/loader.h"


/**

 * Plan a patch at the specified offset within the binary's executable code.

 *

 * @param binary The loaded ELF binary representation

 * @param offset File offset within .text section to patch

 * @param new_bytes The new instruction bytes to insert

 * @param new_size Size of new_bytes in bytes

 * @return Pointer to patch_site_t containing the plan, or NULL on error

 *

 * @note The caller must specify the size of original instructions being replaced.

 * For Milestone 1, we assume caller knows this. Future versions could use

 * disassembly to determine instruction boundaries automatically.

 */

patch_site_t* plan_patch(elf_binary_t* binary, uint64_t offset,

                         unsigned char* new_bytes, size_t new_size) {

    // TODO 1: Validate that offset is within an executable section (.text)

    //   - Use elf_find_section(binary, ".text") to get text section

    //   - Check if offset is between shdr->sh_offset and shdr->sh_offset + shdr->sh_size

    //   - If not, return NULL (or set error)



    // TODO 2: Determine size of original instructions being replaced

    //   - For now, require caller to specify via separate parameter (not in this signature)

    //   - Future: Use disassembler to find instruction boundaries

    //   - For skeleton, assume a fixed original_size (e.g., 5 bytes for typical prologue)

    //   - This is a TEMPORARY simplification for Milestone 1

    size_t original_size = 5; // Placeholder



    // TODO 3: Check if new_size <= original_size (for NOP-padded approach)

    //   - If new_size > original_size, return NULL (can't expand in-place yet)


```

```
    // TODO 4: Calculate padding needed
    size_t padding_needed = original_size - new_size;

    // TODO 5: Allocate and populate patch_site_t structure
    patch_site_t* patch = (patch_site_t*)malloc(sizeof(patch_site_t));
    if (!patch) return NULL;

    // TODO 6: Copy new_bytes into patch->new_bytes (allocate and memcpy)

    // TODO 7: Generate NOP padding if needed using generate_nop_sled()

    // TODO 8: Set other patch fields: target_offset, original_size, new_size,
    //         nop_padding_size, applied=0

    // TODO 9: Return the completed patch plan

    return patch; // Placeholder
}

/**
 * Apply a previously planned patch to the binary's in-memory representation.
 *
 * @param binary The loaded ELF binary to modify
 * @param patch The patch plan to execute
 * @return ERR_SUCCESS on success, error code on failure
 */
int apply_patch(elf_binary_t* binary, patch_site_t* patch) {
    // TODO 1: Validate preconditions
    //   - Check patch != NULL and patch->applied == 0
    //   - Check binary != NULL and binary->buffer != NULL
    //   - Check target_offset is within binary buffer bounds

    // TODO 2: Calculate write address in binary buffer
    //   - write_ptr = binary->buffer + patch->target_offset

    // TODO 3: Write new_bytes to the target location
```

```
    //   - memcpy(write_ptr, patch->new_bytes, patch->new_size)

    // TODO 4: Write NOP padding if present
    //   - if (patch->nop_padding_size > 0)
    //       memcpy(write_ptr + patch->new_size, patch->nop_padding, patch->nop_padding_size)

    // TODO 5: Mark patch as applied
    //   - patch->applied = 1

    // TODO 6: Return success code
    return ERR_SUCCESS; // Placeholder
}

/**
 * Free all resources associated with a patch site.
 *
 * @param patch The patch to deallocate
 */
void patch_free(patch_site_t* patch) {
    if (!patch) return;

    // TODO 1: Free new_bytes if it was allocated by plan_patch
    //   - free(patch->new_bytes)

    // TODO 2: Free nop_padding if present
    //   - free(patch->nop_padding)

    // TODO 3: Free the patch structure itself
    //   - free(patch)
}
```

File: `src/patch/planner.h` (Header)

```c
#ifndef PATCH_PLANNER_H

#define PATCH_PLANNER_H


#include <stdint.h>

#include <stddef.h>

#include "elf/loader.h"


// Patch site structure (already defined in naming conventions, included for completeness)

typedef struct patch_site {

    uint64_t target_offset;

    size_t original_size;

    unsigned char* new_bytes;

    size_t new_size;

    unsigned char* nop_padding;

    size_t nop_padding_size;

    int applied;

} patch_site_t;


// Function declarations

patch_site_t* plan_patch(elf_binary_t* binary, uint64_t offset,

                         unsigned char* new_bytes, size_t new_size);

int apply_patch(elf_binary_t* binary, patch_site_t* patch);

unsigned char* generate_nop_sled(size_t count);

void patch_free(patch_site_t* patch);


#endif // PATCH_PLANNER_H
```

E. Language-Specific Hints (C)

1. **Memory Management**: The C implementation requires careful manual memory management. Follow the convention that functions allocating memory (like `plan_patch`) document who is responsible for freeing it.
2. **Error Handling**: Use the predefined `ERR_*` constants for consistent error reporting. Consider adding an `error_t` parameter to functions for detailed error messages.
3. **Endianness**: When reading/writing multi-byte values in the ELF buffer, ensure you respect the binary's endianness (from `e_ident[EI_DATA]`). Use helper functions like `elf_read_word` that handle endian conversion.
4. **Alignment**: When calculating offsets, respect section alignment requirements (`sh_addralign`). Patching doesn't change alignment but should maintain it.

F. Milestone Checkpoint

After implementing the Patch Planner and Binary Patcher (Milestone 1), verify functionality with these tests:

Test 1: Simple Instruction Replacement

```bash
# Create a test binary with known code
echo -n -e '\x55\x48\x89\xe5\xb8\x2a\x00\x00\x00\xc9\xc3' > test.bin
# (This is a minimal function: push rbp; mov rbp, rsp; mov eax, 0x2a; leave; ret)

# Run your tool to patch the mov eax, 0x2a (bytes b8 2a 00 00 00 at offset 3)
# with xor eax, eax (bytes 31 c0) plus 3 NOPs
./elflex patch --offset 3 --new-bytes 31c0 --original-size 5 test.bin test_patched.bin

# Verify the patch
hexdump -C test_patched.bin
# Should show: 55 48 89 e5 31 c0 90 90 90 c9 c3
```

Test 2: Validation of Error Conditions

```bash
# Attempt to patch outside .text section
./elflex patch --offset 0x1000 --new-bytes 90 --original-size 1 test.bin
# Should fail with "invalid offset" or similar error

# Attempt to patch with larger new bytes
./elflex patch --offset 3 --new-bytes $(printf '%0.20s' '90909090909090909090') --original-size 5 test.bin
# Should fail with "patch would overflow" for Milestone 1
```

Test 3: Round-Trip Execution Test

```bash
# Create a simple C program that prints a value
cat > test_prog.c << 'EOF'
#include <stdio.h>
int main() {
    int x = 42;
    printf("Value: %d\n", x);
    return 0;
}
EOF
gcc -o test_prog test_prog.c

# Patch it to change the constant 42 to 0
# First find the instruction: objdump -d test_prog | grep -A2 "<main>"
# Then use your tool to patch the appropriate mov instruction

# Run the patched program
./test_prog_patched
# Should print "Value: 0" instead of "Value: 42"
```

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Segmentation fault after patching	1. Patch at wrong offset corrupting valid instruction 2. NOP padding overwriting following instructions	Use `objdump -d` on original and patched binary to see instruction boundaries. Check if patched region aligns with function starts.	Ensure patches start at instruction boundaries. Use disassembler in `plan_patch`.
Patched binary won't execute	ELF headers corrupted during patching	Run `readelf -l patched.bin` and compare with original. Check segment permissions, offsets.	Ensure `apply_patch` only modifies `.text` content, not headers. Use `binary_write` from ELF Loader to safely write changes.
Hook not called after patching	Patch bytes incorrect or at wrong location	Use `gdb` to single-step through patched function. Check if first instruction is your patch.	Verify offset calculation: file offset vs virtual address. Use `objdump -h` to map between them.
Partial patch application	Memory allocation failure in `plan_patch`	Add debug prints to track allocation. Check return values of `malloc`, `memcpy`.	Implement proper error checking. Free allocated memory on failure paths.

Component: Trampoline Generator

> **Milestone(s):** 2 (Function Hooking with Trampolines)

The **Trampoline Generator** is the component responsible for creating the code sequences that intercept function calls, enabling custom monitoring or behavior modification. It transforms a simple instruction patch into a sophisticated redirection mechanism that preserves the original function's behavior while inserting new logic. This component is the heart of the instrumentation tool's ability to hook functions without breaking the target binary's execution flow.

Mental Model: The Detour and Callback Service

Imagine a major highway (the program's normal execution flow) with an important exit (a function call). We want to redirect traffic through a service station (our hook function) without causing a permanent roadblock or changing the final destination. The trampoline is precisely this detour system:

1. **Detour Sign (Initial Jump):** The first few bytes of the original function are replaced with a jump that redirects execution off the main highway.
2. **Service Station (Trampoline Code):** This is a small, specially constructed code area that:
   - Provides temporary parking (saves CPU register state)
   - Allows passengers to visit amenities (calls the hook function)
   - Returns passengers to their vehicles (restores registers)
   - Gets traffic back on the original route (executes the original function prologue that was displaced)
3. **Rejoin Highway (Jump to Original):** After the service station visit, traffic continues along the original route to the function's main body.

The critical insight is that the **trampoline must be completely transparent**—after the hook executes, the original function should behave exactly as if nothing happened, except for any intentional modifications the hook made to function arguments or global state. This transparency requires meticulous attention to register preservation, stack management, and instruction relocation.

Interface and Responsibilities

The Trampoline Generator exposes a minimal interface focused on creation and management of trampoline code blocks. Its primary responsibility is to generate position-independent code sequences that can be injected into the binary (either into a new section or into an existing code cave).

Method	Parameters	Returns	Description
`generate_trampoline`	`target_addr uint64_t` (address of function to hook), `hook_addr uint64_t` (address of hook function)	`trampoline_t*` (newly allocated trampoline structure)	Creates a complete trampoline for hooking the function at `target_addr`. The trampoline will call `hook_addr` before jumping to the original function. This includes generating the relocated prologue and the full trampoline code sequence.
`relocate_prologue`	`original_code unsigned char*` (pointer to original function bytes), `size size_t` (number of bytes to relocate)	`unsigned char*` (allocated buffer with relocated instructions)	Copies and potentially modifies the first `size` bytes of the original function so they can be safely executed at a different address. This handles instruction fixing, particularly for RIP-relative addressing modes.
`fixup_rip_relative`	`instruction unsigned char*` (pointer to instruction to fix), `old_ip uint64_t` (original instruction pointer value), `new_ip uint64_t` (new instruction pointer value)	`unsigned char*` (pointer to fixed instruction buffer)	Adjusts a single instruction that uses RIP-relative addressing (common on x86-64 for accessing data and some jumps) to work correctly when executed from a different address. Returns a newly allocated buffer with the fixed instruction.
`trampoline_free`	`trampoline trampoline_t*` (trampoline to deallocate)	`void`	Releases all memory associated with a trampoline structure, including its code buffers and internal data.

The component's internal responsibilities include:

- Determining the optimal size of the original function prologue to replace (typically 5-14 bytes for a jump)
- Generating assembly code for register preservation (push instructions)
- Generating the call to the hook function with proper calling convention
- Generating register restoration (pop instructions)

- Creating the final jump from the trampoline back to the original function (after the replaced prologue)

Internal Behavior: Trampoline Assembly

The trampoline generation follows a precise, multi-step algorithm that produces architecture-specific machine code (x86-64 in our implementation). Below is the detailed procedure:

1. **Analyze Target Function Prologue:**
   - Read the first N bytes from the target function's starting address (where N is at least 5 bytes for a relative jump, but ideally enough to capture complete instructions).
   - Disassemble these bytes to identify instruction boundaries. The goal is to replace a set of whole instructions whose total size is at least 5 bytes (for a 32-bit relative `JMP`).
   - Store these original bytes in `trampoline_t.original_prologue` with their size in `prologue_size`.
2. **Relocate the Prologue:**
   - Copy the original prologue instructions to a new buffer (`relocated_prologue`).
   - For each instruction in the buffer:
     - If the instruction uses **RIP-relative addressing** (common for `MOV` from global data, `CALL` to nearby functions, or certain conditional jumps), calculate the new offset.
     - The fixup formula: `new_displacement = old_displacement + (old_ip - new_ip)`
     - Where `old_ip` is the address where the instruction originally executed (target function address + offset within prologue), and `new_ip` is the address where the relocated instruction will execute (trampoline address + corresponding offset).
     - Update the instruction's displacement field accordingly.
   - Store the fixed buffer in `trampoline_t.relocated_prologue` with its size (same as original prologue size).
3. **Generate Trampoline Code Sequence:** The trampoline is a contiguous block of machine code with the following layout, emitted in order:

   a. **Register Save (Prologue):**
   - Push all general-purpose registers that may be clobbered by the hook or by the trampoline itself onto the stack. On x86-64, this typically means `RAX`, `RCX`, `RDX`, `R8`, `R9`, `R10`, `R11` (scratch registers per System V ABI), and potentially others depending on calling convention.
   - The stack must remain 16-byte aligned per the System V ABI before the `call` instruction.

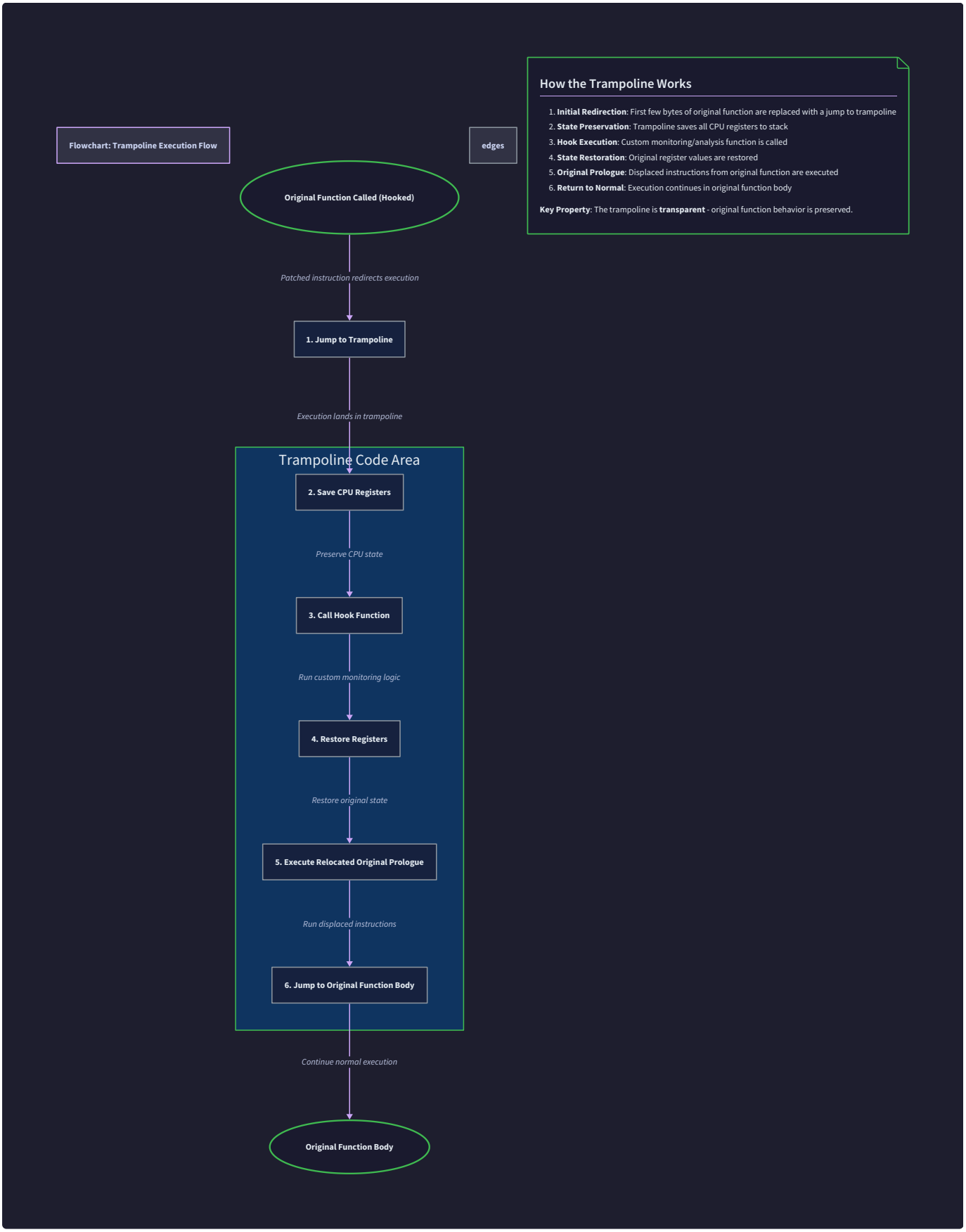
   b. **Hook Function Call:**
   - Load any required arguments (the original function's arguments are already in the correct registers per calling convention).
   - Emit a `call` instruction to the hook function address. This address must be known (either absolute if within ±2GB range, or via register-indirect if not).
   - The hook function can inspect and modify the arguments (which are in registers/stack as per normal calling convention).

   c. **Register Restore (Epilogue):**
   - Pop the saved registers in reverse order to restore the original CPU state.

   d. **Execute Relocated Prologue:**
   - Copy the relocated prologue instructions (from step 2) directly into the trampoline code stream. These instructions will set up the stack frame and perform any other operations the original function prologue did.

   e. **Jump to Original Function Continuation:**
   - Calculate the address of the instruction immediately after the replaced prologue in the original function: `continuation_addr = target_addr + prologue_size`.
   - Emit a `JMP` instruction to this continuation address. Because this jump may exceed the ±2GB range of a relative jump, we use an **indirect jump** via a register (see ADR below).
4. **Finalize Trampoline Structure:**
   - Concatenate all emitted code into a single buffer (`trampoline_t.code`).

- Record the trampoline's size (`code_size`), the hook address, and the target function address in the structure.
- The trampoline is now ready for injection into the binary (via the Code Injector component) or for direct patching into a running process (via the Runtime Patcher).



ADR: Direct Jump vs. Indirect Jump for Trampoline

> **Decision: Use Indirect Jump for Final Trampoline Jump**
>
> - **Context:** After executing the hook and the relocated prologue, the trampoline must transfer control to the original function body (the continuation address). The distance between the trampoline location and the target continuation address can be arbitrarily large (especially if the trampoline is placed in a newly injected section far from the original `.text` segment).
> - **Options Considered:**
>   1. **Direct Relative Jump (`JMP rel32`):** Uses a 32-bit signed offset relative to the next instruction. Simple and compact (5 bytes), but limited to ±2GB range.
>   2. **Indirect Jump via Register (`JMP r64`):** Loads the target address into a register and jumps via the register. Requires extra instructions to load the address, but works for any 64-bit address.
>   3. **Direct Absolute Jump (`JMP imm64`):** Some assemblers support a 64-bit absolute immediate jump, but this is not standard in x86-64—it would require a 14-byte sequence (including REX prefixes) and is less efficient.
> - **Decision:** Use an indirect jump via a register (specifically `R11` , which we already save/restore) for the final jump from trampoline to original function continuation.
> - **Rationale:** The primary concern is reliability across diverse binaries. We cannot guarantee the trampoline will be placed within 2GB of the target function, especially when injecting new sections that may be mapped at arbitrary addresses by the OS loader. The indirect jump, while slightly larger (requires a `MOV` to load the address plus the `JMP`), guarantees correctness regardless of distance. The performance impact is negligible for instrumentation use cases.
> - **Consequences:** The trampoline code size increases by approximately 7-10 bytes (for loading the 64-bit immediate address into a register and the indirect jump). We must ensure the chosen register (`R11`) is properly saved and restored in the trampoline's prologue/epilogue. The absolute address of the continuation must be calculated at trampoline generation time (or patched later during injection if the trampoline's load address isn't known).

Option	Pros	Cons	Chosen?
Direct Relative Jump	Compact (5 bytes), fast execution	Limited to ±2GB range; may fail if trampoline placed far from target	No
Indirect Jump via Register	Works for any distance, reliable	Larger (7-10 bytes), uses one register	**Yes**
Direct Absolute Jump	Conceptually simple	Non-standard encoding, very large (14 bytes), not supported by all assemblers	No

Common Pitfalls

⚠️ Pitfall: Incomplete Register Save/Restore

- **Description:** Failing to save and restore all registers that the trampoline or hook function modifies can corrupt the original function's state, leading to unpredictable crashes or silent data corruption.
- **Why it's wrong:** The original function expects its arguments and scratch registers to be in specific states when it begins execution. If the hook changes `RAX` (which might hold a function argument or be used for other purposes), the original function will receive incorrect data. Similarly, the System V ABI designates certain registers as caller-saved and others as callee-saved; the trampoline must respect this.
- **Fix:** Save all general-purpose registers that are potentially clobbered. A conservative approach is to save `RAX` , `RCX` , `RDX` , `R8` , `R9` , `R10` , `R11` (scratch registers) and also `RDI` , `RSI` , `RBX` , `RBP` , `R12` - `R15` if the hook might modify them (or if you want to be absolutely safe). Restore them in exact reverse order before executing the relocated prologue. Use `push` / `pop` instructions or a stack frame.

⚠️ Pitfall: Incorrect RIP-Relative Fixup

- **Description:** When relocating the original prologue, instructions that use RIP-relative addressing (like `mov rax, [rip+0x1234]` or `call rip+0x5678`) are not adjusted for their new location, causing them to access wrong memory addresses or jump to wrong code locations.
- **Why it's wrong:** RIP-relative addressing calculates the effective address based on the current instruction pointer. If you move the instruction without adjusting the offset, it will point to a location relative to the new IP, which is incorrect for the original intent (e.g., accessing a global variable at a fixed absolute address).

- **Fix:** Implement a disassembler-assisted fixup routine. For each relocated instruction, check if it uses RIP-relative addressing. If so, compute the original target address: `original_target = old_ip + instruction_length + displacement`. Then compute the new displacement: `new_displacement = original_target - (new_ip + instruction_length)`. Update the instruction's displacement field. This requires accurate instruction length decoding (use a library like `libcapstone` or `Zydis`).

⚠️ Pitfall: Stack Misalignment

- **Description:** The x86-64 System V ABI requires the stack to be 16-byte aligned when a `call` instruction is executed. If the trampoline's `call hook` occurs with a misaligned stack, it can cause crashes in the hook function (especially if it uses SSE instructions or calls further functions).
- **Why it's wrong:** The original function's prologue may have pushed a return address (8 bytes) and possibly other registers, changing stack alignment. The trampoline adds its own pushes, which must be accounted for to maintain 16-byte alignment before the `call`.
- **Fix:** Count the number of bytes pushed on the stack (each `push` is 8 bytes). Ensure that `(original_stack_pointer - num_pushed_bytes) % 16 == 8` before the `call` (because the `call` itself pushes an 8-byte return address, making the stack 16-byte aligned at the callee's entry). If needed, add an extra `sub rsp, 8` or similar to adjust.

⚠️ Pitfall: Assuming Fixed Prologue Size

- **Description:** Hardcoding the number of bytes to replace (e.g., always replacing 5 bytes) without ensuring those bytes represent complete instructions can split an instruction, leading to invalid opcodes and crashes.
- **Why it's wrong:** Functions may start with instructions of varying lengths (1–15 bytes). Replacing only part of an instruction leaves the remaining bytes to be interpreted as opcodes when execution later jumps to the relocated prologue, causing unpredictable behavior.
- **Fix:** Use a disassembler to iterate byte-by-byte from the function start, accumulating instructions until the total size is at least the required jump size (5 bytes for `JMP rel32`). Ensure the last instruction is completely included. For safety, add a small buffer (e.g., target at least 12–14 bytes) to accommodate common prologues and avoid pathological cases.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Disassembly for prologue analysis	Manual instruction decoding (pattern matching) for common instructions	Use `capstone` or `Zydis` library for robust disassembly
Instruction modification	Hardcode common RIP-relative instruction patterns (e.g., `0xFF 0x15` for `call [rip+disp]`)	Full instruction decoder/encoder
Code emission	Inline assembly macros or manually encoded byte arrays	Use `libjit` or `dynasm` for dynamic code generation

B. Recommended File/Module Structure:

```
elflex/
├── src/
│   ├── elf/                # ELF parsing components
│   ├── patch/              # Patch planner and patcher
│   ├── trampoline/         # Trampoline generator component (this section)
│   │   ├── trampoline.c    # Main trampoline generation logic
│   │   ├── trampoline.h    # Public interface
│   │   ├── reloc.c         # Prologue relocation and RIP fixup
│   │   └── arch_x64.c      # x86-64 specific code emission
│   ├── inject/             # Code injection component
│   └── runtime/            # Runtime ptrace component
└── tools/                  # Test binaries and utilities
```

C. Infrastructure Starter Code: Complete helper for encoding common x86-64 instructions (to be placed in `arch_x64.c`):

```c
// arch_x64.c - x86-64 instruction encoding helpers

#include <stdint.h>

#include <stdlib.h>

#include <string.h>


// Encode a push register instruction (0x50 + reg_code)

unsigned char* encode_push_reg(int reg_code, unsigned char* buf) {

    buf[0] = 0x50 + (reg_code & 7);

    if (reg_code >= 8) {

        // Need REX prefix for R8-R15

        buf[0] = 0x41;

        buf[1] = 0x50 + (reg_code & 7);

        return buf + 2;

    }

    return buf + 1;

}


// Encode a pop register instruction (0x58 + reg_code)

unsigned char* encode_pop_reg(int reg_code, unsigned char* buf) {

    buf[0] = 0x58 + (reg_code & 7);

    if (reg_code >= 8) {

        buf[0] = 0x41;

        buf[1] = 0x58 + (reg_code & 7);

        return buf + 2;

    }

    return buf + 1;

}


// Encode a near call with 32-bit relative offset (E8 cd)

unsigned char* encode_call_rel32(uint32_t offset, unsigned char* buf) {

    buf[0] = 0xE8;

    *(uint32_t*)(buf + 1) = offset;

    return buf + 5;

}


// Encode a near jump indirect via register (FF /4, modrm: 11 100 reg)

unsigned char* encode_jmp_reg(int reg_code, unsigned char* buf) {

```

```
    buf[0] = 0xFF;
    buf[1] = 0xE0 + (reg_code & 7);
    if (reg_code >= 8) {
        // Need REX.R prefix bit
        buf[0] = 0x41;
        buf[1] = 0xFF;
        buf[2] = 0xE0 + (reg_code & 7);
        return buf + 3;
    }
    return buf + 2;
}

// Encode mov immediate 64-bit to register (48 B8+rd io)
unsigned char* encode_mov_imm64_to_reg(int reg_code, uint64_t value, unsigned char* buf) {
    // REX.W prefix (0x48) for 64-bit operand
    buf[0] = 0x48 + ((reg_code >= 8) ? 0x04 : 0x00); // REX.W | REX.R if reg >= 8
    buf[1] = 0xB8 + (reg_code & 7);
    *(uint64_t*)(buf + 2) = value;
    return buf + 10;
}
```

D. Core Logic Skeleton Code: Trampoline generation main function (to be placed in `trampoline.c`):

```c
// trampoline.c - Core trampoline generation logic

#include "trampoline.h"

#include <stdlib.h>

#include <string.h>


// Generate a trampoline for hooking target function

trampoline_t* generate_trampoline(uint64_t target_addr, uint64_t hook_addr) {

    // TODO 1: Allocate and initialize trampoline_t structure

    //   - Use calloc to allocate trampoline_t

    //   - Set target_function_address and hook_address fields



    // TODO 2: Determine prologue size to replace

    //   - Read first N bytes from target_addr (you'll need access to binary)

    //   - Disassemble until you have at least 5 bytes of complete instructions

    //   - Store original prologue bytes and size in trampoline_t



    // TODO 3: Relocate the prologue

    //   - Call relocate_prologue(original_prologue, prologue_size)

    //   - Store result in trampoline_t.relocated_prologue



    // TODO 4: Calculate addresses

    //   - continuation_addr = target_addr + prologue_size

    //   - For now, assume trampoline_address is unknown (0); will be filled during injection



    // TODO 5: Generate trampoline code

    //   - Estimate total code size (register save/restore + call + relocated prologue + jump)

    //   - Allocate buffer for code

    //   - Emit register save (push RAX, RCX, RDX, R8, R9, R10, R11)

    //   - Ensure stack alignment (adjust if needed)

    //   - Emit call to hook_addr (relative call if within range, otherwise indirect)

    //   - Emit register restore (pop in reverse order)

    //   - Copy relocated_prologue into buffer

    //   - Emit mov R11, continuation_addr

    //   - Emit jmp R11

    //   - Store final code and size in trampoline_t


```

```
    // TODO 6: Return the trampoline structure
    return NULL; // placeholder
}

// Relocate original function prologue, fixing RIP-relative instructions
unsigned char* relocate_prologue(unsigned char* original_code, size_t size) {
    // TODO 1: Allocate buffer for relocated code (same size as input)


    // TODO 2: Copy original code to buffer


    // TODO 3: For each instruction in the buffer:
    //   - Decode instruction length (simple pattern matching or use disassembler)
    //   - Check if instruction uses RIP-relative addressing
    //   - Common patterns:
    //       * E8 cd: CALL rel32 (relative to next instruction)
    //       * E9 cd: JMP rel32
    //       * 0F 84 cd: JE rel32
    //       * 48 8B 05 rd: MOV RAX, [RIP+disp32] (and other variants)
    //       * FF 15 rd: CALL [RIP+disp32]
    //       * FF 25 rd: JMP [RIP+disp32]
    //   - For RIP-relative instructions:
    //       * Extract displacement from instruction (varies by opcode)
    //       * old_ip = original_location_of_instruction (you need to know this)
    //       * new_ip = relocated_location_of_instruction
    //       * original_target = old_ip + instruction_length + displacement
    //       * new_displacement = original_target - (new_ip + instruction_length)
    //       * Update displacement in the relocated instruction


    // TODO 4: Return relocated buffer
    return NULL; // placeholder
}

// Fix a single RIP-relative instruction (simplified interface)
unsigned char* fixup_rip_relative(unsigned char* instruction, uint64_t old_ip, uint64_t new_ip) {
    // TODO 1: Copy instruction to new buffer
```

```
    // TODO 2: Determine instruction length and displacement offset based on opcode


    // TODO 3: Calculate original_target = old_ip + length + displacement


    // TODO 4: Calculate new_displacement = original_target - (new_ip + length)


    // TODO 5: Update displacement in the copied instruction


    // TODO 6: Return the fixed instruction buffer
    return NULL; // placeholder
}
```

E. Language-Specific Hints:

- **Memory Management:** The trampoline generator allocates multiple buffers (`original_prologue`, `relocated_prologue`, `code`). Ensure `trampoline_free()` deallocates all of them to prevent memory leaks.
- **Instruction Encoding:** x86-64 instructions have variable length and complex encoding. Start with a minimal set of supported instructions (e.g., `push`, `pop`, `call`, `mov`, `jmp`) for the trampoline itself. For prologue relocation, focus on the most common RIP-relative patterns found in function prologues.
- **Alignment:** Use `memalign()` or `posix_memalign()` to allocate trampoline code buffers with 16-byte alignment, as some processors may favor aligned code.
- **Testing:** Create simple test functions with known prologues (e.g., `push rbp; mov rbp, rsp`) to verify trampoline generation works before testing on real binaries.

F. Milestone Checkpoint: After implementing the trampoline generator, test it with a simple target function:

1. Create a test binary with a function `int add(int a, int b) { return a + b; }`
2. Generate a trampoline for `add` with a dummy hook address (e.g., `0x1000`).
3. Inspect the generated trampoline code with `objdump -D -b binary -m i386:x86-64 <trampoline.bin>`
4. Verify the trampoline contains:
    - Push instructions saving registers
    - A call to the hook address
    - Pop instructions restoring registers
    - The original prologue instructions
    - A jump to the continuation address
5. Expected output: A valid x86-64 code sequence that can be disassembled without errors.

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Hook is called but original function crashes	Incorrect register save/restore or stack misalignment	Single-step through trampoline in GDB, check register values before/after hook call	Verify push/pop count matches, ensure 16-byte stack alignment before `call`
Original function executes but with wrong arguments	Hook corrupted argument registers	Check which registers the hook modifies; ensure trampoline saves/restores all argument registers (RDI, RSI, RDX, RCX, R8, R9)	Save all argument registers in trampoline prologue
Crash when executing relocated prologue	RIP-relative instructions not fixed	Disassemble relocated prologue and compare with original; check if `[RIP+...]` addresses are correct	Implement proper `fixup_rip_relative` for all common RIP-relative instruction forms
Trampoline works in static binary but not when injected	Absolute addresses not adjusted for load address	The trampoline contains absolute addresses (e.g., `mov R11, 0x...`) that assume a specific load address	Either generate position-independent trampolines or apply relocations during injection (see Code Injector component)

Component: Code Injector and Section Manager

> **Milestone(s):** 3 (Code Injection)

The **Code Injector and Section Manager** is the architectural component responsible for extending the ELF binary's structure by adding entirely new executable sections containing custom hook code. While the Patch Planner handles modifications within existing sections and the Trampoline Generator creates redirection stubs, this component fundamentally alters the binary's layout by introducing new code regions that become part of the executable's address space. This enables the injection of complex monitoring logic, data collection routines, or even entirely new functionality into a compiled binary without access to its source code.

Mental Model: The Building Annex

Think of a compiled ELF binary as a completed building with carefully planned rooms (sections) arranged on specific floors (virtual address ranges). The original architects (compilers) designed each room's purpose: `.text` for living quarters (code), `.data` for storage (initialized data), and so on. The **Code Injector** is like a renovation team tasked with adding an entirely new annex to this building while it remains occupied and functional.

The challenge has three layers:

1. **Blueprint Modification (ELF Headers)**: Just as adding an annex requires updating architectural blueprints, injecting code requires modifying ELF headers to declare the new section's existence, location, and permissions. The injector must create new section headers and potentially new program headers (segments) that tell the operating system loader to map this new region into memory with executable permissions.
2. **Physical Construction (Section Data)**: The annex itself—the actual injected code—must be constructed as position-independent "prefabricated modules" that can function correctly regardless of where in the address space they're placed. This code must be compiled separately and then inserted into the binary file at the correct offset.
3. **Utilities Connection (Symbol Resolution)**: The new annex needs access to the building's utilities (existing functions and data). The injector must resolve references from the injected code to symbols in the original binary, similar to connecting the annex to the main building's electrical and plumbing systems. This involves calculating correct addresses for external function calls and potentially updating relocation tables.

The entire operation must maintain **ELF structural integrity**: shifting existing sections to make room (or expanding into empty space), updating all references to moved content, and ensuring the final binary remains loadable by the OS. The result is a seamless integration where trampolines in the original `.text` section can jump to the new `.inject` section, which in turn can call back to original functions, creating a symbiotic relationship between original and injected code.

Interface and Responsibilities

The Code Injector exposes a focused API for extending the binary with new executable content while managing the complex header updates and address resolution required. Its primary consumer is the main orchestration logic that combines trampoline generation with code injection to implement complete function hooking.

Method	Parameters	Returns	Description
`inject_section`	`binary elf_binary_t*`, `name char*`, `data unsigned char*`, `size size_t`, `flags uint64_t`	`int` (error code)	Primary entry point for section injection. Adds a new section with given name, content, and flags (e.g., `SHF_ALLOC | SHF_EXECINSTR`). Handles all header updates, address assignment, and file layout adjustments. Returns `ERR_SUCCESS` on success or appropriate error code.
`allocate_code_cave`	`binary elf_binary_t*`, `size size_t`	`uint64_t` (virtual address)	Scans existing segments for unused padding ("code caves") large enough to hold the requested size. Returns the virtual address of the cave, or 0 if none found. Used as an alternative to adding new segments when space exists within existing memory regions.
`resolve_symbol_in_target`	`binary elf_binary_t*`, `symbol_name char*`	`uint64_t` (address)	Looks up a symbol by name in the target binary's symbol table and returns its virtual address. Essential for injected code that needs to call original functions. Returns 0 if symbol not found.
`injected_section_free`	`section injected_section_t*`	`void`	Deallocates memory for an `injected_section_t` structure, including its name and data buffers. Called during cleanup phases.

The component also relies on several internal helper functions not exposed in the public API:

- `expand_elf_for_section()`: Increases file size and shifts subsequent sections to create space
- `create_new_program_header()`: Allocates and initializes a new `PT_LOAD` segment for injected code
- `apply_relocations_to_injected_code()`: Patches address references within injected code based on final load addresses
- `update_all_offsets()`: Recursively adjusts file offsets in all headers after insertion point

Internal Behavior: Section Injection Algorithm

The injection process follows a precise, multi-stage algorithm that transforms the in-memory ELF representation while maintaining all invariants. Consider injecting a hook function compiled to position-independent code that monitors calls to `printf`:

1. **Validation and Planning Phase**
    - Validate that the binary has sufficient header space for an additional section header (expand section header table if needed)
    - Determine the injection strategy: add new `PT_LOAD` segment or use existing segment padding (see ADR below)
    - If adding new segment, calculate a virtual address range that doesn't conflict with existing segments and maintains proper alignment (typically page-aligned, 0x1000 bytes)
2. **Space Allocation Phase**
    - **If using new segment**: Calculate required file space by aligning injection size to section alignment (often 0x10). Determine insertion point—typically at end of file for simplicity.
    - **If reusing code cave**: Verify the cave is within an existing executable segment and has sufficient size. Ensure cave boundaries don't split existing sections.
    - Allocate temporary buffer for modified binary content (original size + injection size + padding)
3. **Header Creation and Update Phase**
    - Create new `Elf64_Shdr` for the injected section:
        - `sh_name`: Index into section string table (add name if not present)

        - `sh_type`: `SHT_PROGBITS` (contains actual code/data)
        - `sh_flags`: `SHF_ALLOC | SHF_EXECINSTR` (allocated in memory, executable)
        - `sh_addr`: Assigned virtual address (from step 1)
        - `sh_offset`: File offset where injected code will reside
        - `sh_size`: Size of injected code
        - Other fields: Set to typical values (alignment = 0x10, `sh_link` = 0, etc.)
    - If creating new segment, also create `Elf64_Phdr`:
        - `p_type`: `PT_LOAD`
        - `p_flags`: `PF_R | PF_X` (readable and executable)
        - `p_offset`: File offset (aligned to page size)
        - `p_vaddr`: Virtual address (same as `sh_addr`, page-aligned)
        - `p_paddr`: Typically same as `p_vaddr`
        - `p_filesz`: Size in file (may include zero-padding to page boundary)
        - `p_memsz`: Same as `p_filesz` (no .bss-like uninitialized data)
        - `p_align`: 0x1000 (4KB page alignment)
    - Update ELF header: Increment `e_shnum` (section count), possibly `e_phnum` (program header count)
    - Update all subsequent section headers: Increase their `sh_offset` values by insertion size if injecting before them
4. **Code Preparation and Relocation Phase**
    - The injected code must be **position-independent code (PIC)** or must be relocated:
        - If PIC: Ensure all memory references use RIP-relative addressing or GOT/PLT indirection
        - If not PIC: Scan code for absolute address references and adjust by adding base offset
    - Resolve external references: For each symbol the injected code needs (e.g., `printf`), call `resolve_symbol_in_target()` and patch the call instruction with correct address
    - Apply any necessary relocations if the injected code has its own relocation table (advanced scenario)
5. **Data Integration Phase**
    - Copy original binary content up to insertion point to new buffer
    - Insert injected code at calculated offset
    - Copy remaining original content after insertion point, adjusting any internal file offset references if sections were shifted
    - Update the `elf_binary_t` structure's internal buffers and metadata to reflect new layout
6. **Verification Phase**
    - Run internal consistency checks: verify no segment overlaps in virtual address space, all offsets are within file bounds, section alignment requirements satisfied
    - Update internal data structures: Add new `section_t` to `elf_binary_t->sections` array, update `segment_t` array if new segment created
    - Return success; binary is now ready for `binary_write()` to produce final patched file

The algorithm's complexity lies in maintaining **referential integrity**: every file offset reference in headers must remain consistent, and every virtual address reference in code must target the correct location after layout changes.

ADR: New Section vs. Code Cave Utilization

> **Decision: Prefer New Sections Over Code Caves for Primary Injection Method**
>
> - **Context**: Injected code requires virtual address space and file space. Two approaches exist: 1) Find unused gaps ("code caves") within existing segments, or 2) Add entirely new segments/sections. The tool must choose a default strategy that balances reliability, simplicity, and flexibility.
> - **Options Considered**:
>   1. **Primary: New Sections/Segments**: Always add new `PT_LOAD` segments for injected code, placing it at the end of the virtual address space.
>   2. **Fallback: Code Cave Utilization**: Scan existing executable segments for padding between sections or at segment ends, inject code there without adding headers.
>   3. **Hybrid: Cave-First with Fallback**: Prefer code caves for small injections but add segments for larger ones.
> - **Decision**: Implement new section/segment addition as primary method, with code cave detection as optional optimization for specific use cases.
> - **Rationale**:
>   - **Predictability**: New segments have guaranteed, contiguous space without size limitations from existing layout.
>   - **Simpler Accounting**: No need to parse complex padding patterns or worry about unintended side effects from reused padding.
>   - **Clear Separation**: Injected code lives in distinct address ranges, making debugging and analysis easier.
>   - **Alignment Guarantees**: New segments can be page-aligned, ensuring proper memory protection.
>   - **ELF Standard Compliance**: Adding sections/segments is a well-documented ELF operation vs. code caves which are implementation artifacts.
> - **Consequences**:
>   - Increases binary size more than code cave reuse.
>   - May cause address space fragmentation in limited 32-bit environments (less concern for 64-bit).
>   - Requires more complex header manipulation code.
>   - Makes injected code more visible to analysis tools (both advantage and disadvantage).

Option	Pros	Cons	Chosen?
New Sections/Segments	Guaranteed space, clear separation, predictable addresses, standard-compliant	Larger binary size, more complex implementation, visible to analysis	**Primary**
Code Cave Utilization	Minimal size increase, stealthier, uses existing permissions	Unpredictable availability, may fragment code, risk of breaking padding assumptions	Fallback only
Hybrid Approach	Balances size and reliability	Most complex implementation, dual code paths	Rejected for initial implementation

This decision prioritizes implementation clarity and reliability over binary size optimization, which aligns with the educational goals of the project. Learners first master the standard-compliant approach before potentially implementing cave detection as an advanced optimization.

Common Pitfalls

⚠️ Pitfall: Segment Overlap Causing Loader Rejection

- **Description**: When adding a new `PT_LOAD` segment, its virtual address range `[p_vaddr, p_vaddr + p_memsz)` overlaps with an existing segment's range. The OS loader will reject such binaries as malformed.
- **Why it's wrong**: Overlapping segments violate ELF specification, causing runtime failure. Common when incorrectly calculating address ranges or forgetting about alignment padding.
- **How to avoid**: Always scan existing segments before assigning `p_vaddr`. Ensure new segment starts at least one page after the highest existing segment's end address: `new_vaddr = ALIGN_UP(prev_vaddr + prev_memsz, PAGE_SIZE)`.

⚠️ Pitfall: Incorrect File Offset Alignment

- **Description**: Section offset (`sh_offset`) not aligned to `sh_addralign` value (often 0x10), or segment offset (`p_offset`) not aligned to `p_align` (often 0x1000).
- **Why it's wrong**: Misaligned sections cause undefined behavior; misaligned segments may prevent loading entirely. The ELF specification requires these alignments.
- **How to avoid**: Always apply proper alignment: `offset = ALIGN_UP(raw_offset, alignment)` . Use helper functions like `align_up(value, alignment)` throughout injection code.

⚠️ Pitfall: Symbol Table Expansion Without String Table Updates

- **Description**: When the injected section needs a name added to the section header string table (`.shstrtab`), or when injected code adds new symbols to the symbol table (`.symtab`), the corresponding string table (`.strtab`) must also be expanded.
- **Why it's wrong**: Symbol entries reference string table indices; if strings are added without updating indices in existing symbols, or if table growth isn't accounted for, symbol resolution breaks.
- **How to avoid**: Treat `.shstrtab` , `.symtab` , and `.strtab` as a coordinated unit. When adding a section name, append to `.shstrtab` , update its size in header, and use the new index for `sh_name` . Recalculate all existing `sh_name` values if table was shifted in file.

⚠️ Pitfall: Assuming Injected Code is Position-Independent

- **Description**: Injecting code compiled for a fixed address (not PIC) without adjusting absolute address references causes crashes because code executes at different address than compiled for.
- **Why it's wrong**: Absolute addresses in injected code point to wrong memory locations. RIP-relative references may also break if relative distances change.
- **How to avoid**: Always compile hook code with `-fPIC -pie` flags for position-independent code. If PIC isn't possible, implement a relocation pass: scan injected code for address constants and adjust by adding base offset (actual_load_address - expected_load_address).

⚠️ Pitfall: Forgetting to Update `e_shoff` After Header Table Expansion

- **Description**: When the section header table grows (adding new section headers), its location in file may shift. The ELF header's `e_shoff` field must point to the new location.
- **Why it's wrong**: `readelf` and loaders use `e_shoff` to find section headers; incorrect value makes binary unparsable.
- **How to avoid**: After any operation that shifts the section header table's file position, immediately update `binary->ehdr->e_shoff` . Track all header movements in a single "offset delta" variable applied consistently.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Section Management	Manual header manipulation with helper functions	libelf (ELF manipulation library)
Position-Independent Code	GCC/Clang `-fPIC -pie` compilation	Custom relocation patching engine
Symbol Resolution	Linear scan of symbol table	Hash table acceleration, demangling support
Address Space Management	Simple end-of-address-space allocation	Sophisticated gap detection with best-fit

For this educational project, the Simple Options are recommended to maximize understanding of ELF internals.

Recommended File/Module Structure

```
elflex/                          # Project root
├── include/                     # Public headers
│   ├── injector.h               # Code Injector interface
│   └── elf_types.h              # ELF structure definitions
├── src/
│   ├── injector/                # Code Injector component
│   │   ├── injector.c           # Main injection logic
│   │   ├── code_cave.c          # Code cave detection
│   │   ├── symbol_resolve.c     # Symbol resolution
│   │   └── relocation.c         # PIC relocation helpers
│   ├── elf/                     # ELF Loader (previous component)
│   ├── patch/                   # Patch Planner (previous component)
│   ├── trampoline/              # Trampoline Generator (previous component)
│   └── main.c                   # Command-line interface
├── hooks/                       # Example hook code to inject
│   ├── monitor.c                # Sample monitoring hook
│   └── Makefile.hook           # Builds hooks as PIC
└── Makefile                     # Main build system
```

The `injector.h` header should declare the public API functions exactly as specified in the Interface table, while internal helpers are declared only within `injector.c`.

Infrastructure Starter Code

File: `include/elf_types.h` (additional definitions for injection)

```c
/* Add to existing elf_types.h */

#ifndef ELF_TYPES_H

#define ELF_TYPES_H


#include <stdint.h>

#include <stddef.h>


/* ... existing ELF type definitions ... */


/* Injection-specific constants */

#define INJECT_SECTION_NAME ".inject"

#define INJECT_SECTION_FLAGS (SHF_ALLOC | SHF_EXECINSTR)

#define PAGE_SIZE 0x1000

#define SECTION_ALIGNMENT 0x10


/* Helper macro for alignment */

#define ALIGN_UP(x, align) (((x) + (align) - 1) & ~((align) - 1))

#define ALIGN_DOWN(x, align) ((x) & ~((align) - 1))


#endif /* ELF_TYPES_H */
```

File: `src/injector/relocation.c` (PIC address patching helper)

```c
#include <injector.h>
#include <string.h>
#include <stdint.h>

/*
 * Patch an absolute address in injected code.
 * This simplistic approach works for known patterns but real
 * relocation requires proper ELF relocation processing.
 */
void patch_absolute_address(uint8_t *code, size_t code_size,
                            uint64_t patch_offset,
                            uint64_t old_address,
                            uint64_t new_address) {
    /* Scan for 8-byte values matching old_address */
    for (size_t i = 0; i <= code_size - 8; i++) {
        uint64_t *potential_addr = (uint64_t*)(code + i);
        if (*potential_addr == old_address) {
            *potential_addr = new_address;
            /* Note: assumes little-endian; for production code,
               handle endianness properly */
        }
    }
}

/*
 * Check if code appears to be position-independent.
 * Simple heuristic: look for RIP-relative opcodes (0x8d for LEAL)
 * or absence of hardcoded 64-bit immediates.
 */
int is_likely_pic(const uint8_t *code, size_t size) {
    /* This is a simplified check - real detection requires disassembly */
    const uint8_t rip_relative_prefixes[] = {0x8d, 0x48, 0x4c}; /* Common PIC patterns */
    for (size_t i = 0; i < size; i++) {
        for (size_t j = 0; j < sizeof(rip_relative_prefixes); j++) {
            if (code[i] == rip_relative_prefixes[j]) {
                return 1; /* Likely PIC */
```

```
            }
        }
    }
    return 0; /* Cannot guarantee PIC */
}
```

Core Logic Skeleton Code

File: `src/injector/injector.c` **(main injection routine)**

```c
#include <injector.h>
#include <elf_loader.h> /* For elf_binary_t */
#include <stdlib.h>
#include <string.h>
#include <errno.h>

/* Internal helper declarations */
static int expand_elf_for_section(elf_binary_t *binary,
                                  size_t inject_size,
                                  size_t *out_file_offset);
static int create_injected_section_header(elf_binary_t *binary,
                                          const char *name,
                                          uint64_t vaddr,
                                          uint64_t file_offset,
                                          size_t size,
                                          uint64_t flags);
static int create_new_load_segment_if_needed(elf_binary_t *binary,
                                             uint64_t vaddr,
                                             uint64_t file_offset,
                                             size_t size);

/*
 * inject_section - Main entry point for injecting a new section
 * Returns ERR_SUCCESS on success, error code on failure
 */
int inject_section(elf_binary_t *binary, const char *name,
                   uint8_t *data, size_t size, uint64_t flags) {
    // TODO 1: Validate inputs: binary not NULL, name not NULL,
    //         data not NULL if size > 0, flags valid


    // TODO 2: Calculate virtual address for new section
    //   - If adding to existing executable segment: call allocate_code_cave()
    //   - If creating new segment: find highest existing segment end,
    //     align up to PAGE_SIZE, use as base address


    // TODO 3: Allocate file space for the injected data
```

```
    //   - Call expand_elf_for_section() to determine file offset
    //   - This function handles shifting existing content

    // TODO 4: Create new section header
    //   - Call create_injected_section_header() with calculated vaddr,
    //     file_offset, size, flags
    //   - Update section name in .shstrtab if needed

    // TODO 5: Create new program header if needed
    //   - If using new segment: call create_new_load_segment_if_needed()
    //   - Set proper permissions (PF_R | PF_X for executable code)

    // TODO 6: Prepare the injected code data
    //   - If code is not position-independent: apply relocation patches
    //     using patch_absolute_address() helper
    //   - Resolve external symbols: for each external reference in data,
    //     call resolve_symbol_in_target() and patch call sites

    // TODO 7: Integrate data into binary buffer
    //   - Copy data to binary->buffer at calculated file_offset
    //   - Update binary->size to reflect new total size

    // TODO 8: Update internal elf_binary_t structures
    //   - Add new section_t to binary->sections array (realloc if needed)
    //   - Add new segment_t to binary->segments array if created
    //   - Update binary->section_count / binary->segment_count

    // TODO 9: Run consistency checks
    //   - Verify no segment overlaps in virtual address space
    //   - Verify all offsets are within buffer bounds
    //   - Verify alignment requirements satisfied

    return ERR_SUCCESS;
}

/*
```

```
 * allocate_code_cave - Find unused space in existing executable segments
 * Returns virtual address of cave, or 0 if no suitable cave found
 */
uint64_t allocate_code_cave(elf_binary_t *binary, size_t required_size) {
    // TODO 1: Iterate through all segments (binary->segments)

    // TODO 2: For each segment with executable permissions (PF_X):
    //   a) Examine gaps between sections within this segment
    //   b) Check end of segment for padding after last section
    //   c) Look for continuous zero bytes (0x00 or 0x90 NOPs) of required_size

    // TODO 3: Return first suitable virtual address found
    //   - Address = segment_base + offset_within_segment
    //   - Ensure address is properly aligned (0x10 typical)

    // TODO 4: If no cave found, return 0
    return 0;
}

/*
 * resolve_symbol_in_target - Look up symbol address in target binary
 * Returns virtual address, or 0 if symbol not found
 */
uint64_t resolve_symbol_in_target(elf_binary_t *binary,
                                  const char *symbol_name) {
    // TODO 1: Locate symbol table section (binary->symtab)
    //   - If NULL, try finding .symtab or .dynsym by name

    // TODO 2: Locate corresponding string table
    //   - Symbol table's sh_link points to string table section index

    // TODO 3: Linear scan through symbol table entries
    //   - For each Elf64_Sym entry:
    //     a) Get symbol name: string_table + sym->st_name
    //     b) Compare with symbol_name using strcmp()
    //     c) If match, return binary load address + sym->st_value
```

```
    //        (adjust for PIC/PIE if needed)


    // TODO 4: If symbol not found, return 0
    return 0;
}
```

Language-Specific Hints (C)

- **Memory Management**: The injection process allocates new buffers for expanded binary content. Use `realloc()` carefully, ensuring to update all pointers within `elf_binary_t` after reallocation.
- **Pointer Arithmetic**: When calculating offsets in the binary buffer, use `uintptr_t` for safe pointer arithmetic: `new_buffer = original_buffer + insert_offset`.
- **Endianness**: ELF headers use the binary's native endianness. Use helper functions like `elf64_to_cpu()` (if implementing cross-endian support) when reading/writing header fields.
- **Debug Output**: Add extensive logging with `#ifdef DEBUG` blocks to trace injection decisions: chosen addresses, section offsets, symbol resolutions.
- **Error Cleanup**: Implement goto-based cleanup pattern for complex functions:

```c
int inject_section(...) {
    uint8_t *backup = NULL;
    section_t *new_section = NULL;

    backup = malloc(...);
    if (!backup) { result = ERR_OUT_OF_MEMORY; goto cleanup; }

    new_section = malloc(...);
    if (!new_section) { result = ERR_OUT_OF_MEMORY; goto cleanup; }

    // ... main logic ...

cleanup:
    if (result != ERR_SUCCESS) {
        free(backup);
        free(new_section);
        // Roll back any partial changes to binary
    }
    return result;
}
```

Milestone Checkpoint

After implementing the Code Injector, verify functionality with these steps:

1. **Build a test hook**:

```bash
# Compile a simple C function as position-independent code
gcc -fPIC -pie -c -o hook.o hooks/monitor.c
objcopy -O binary --only-section=.text hook.o hook.bin
```

2. **Run injection test**:

```bash
# Use elflex to inject into a simple test binary
./elflex inject test_program --section .inject --hook hook.bin --symbol printf
```

3. **Verify results**:

```bash
# Check that new section appears
readelf -S patched_test_program | grep .inject
# Should show .inject section with flags AX (allocated, executable)

# Check that segment count increased
readelf -l patched_test_program | grep LOAD
# Should show one more PT_LOAD segment than original

# Test execution
./patched_test_program
# Should run with injected monitoring (add printf to hook to see output)
```

Expected Success Indicators:

- Patched binary executes without segmentation faults
- New `.inject` section visible in `readelf` output
- Program behavior unchanged except for added hook functionality

Common Failure Signs:

- `readelf: Error: Not an ELF file` → ELF headers corrupted during injection
- `Segmentation fault` → Injected code not properly relocated or symbol resolution incorrect
- `Killed` (SIGKILL) → Memory permissions wrong or segment overlap causing loader rejection

Component: Runtime Patcher (via ptrace)

> **Milestone(s):** 4 (Runtime Patching via ptrace)

The **Runtime Patcher (via ptrace)** is the component that enables **dynamic binary instrumentation** — modifying running processes in real-time without requiring a restart. This component represents the most advanced capability of ELFlex, allowing instrumentation of long-running services, debugging of production systems, and security analysis of live applications. Unlike static binary rewriting (which modifies files on disk), runtime patching must navigate the complexities of a live process: its memory layout, execution state, and operating system protections.

Mental Model: The Live Surgery Team

Imagine a running process as a patient undergoing open-heart surgery while still conscious and performing their daily functions. The runtime patcher is the **surgical team** that must:

1. **Safely anesthetize the patient** (stop the process with `ptrace`)
2. **Make precise incisions** (modify executable memory pages)
3. **Perform the transplant** (inject trampoline code)
4. **Close up and restart vital functions** (restore memory protections and resume execution)

The challenge is that the "patient" (process) has an immune system: memory pages are marked read-only for code (`PROT_READ|PROT_EXEC`), the instruction cache may hold stale copies of modified code, and other system components (like signal handlers) might interfere. The surgical team uses specialized tools (`ptrace` system calls) to temporarily bypass these protections, make their modifications, and ensure the patient wakes up healthy with the new "implant" (hook) functioning correctly.

This mental model emphasizes three critical constraints:

- **Minimal intervention time**: The longer the process is stopped, the more it disrupts system function (like keeping a patient under anesthesia)
- **Precision and cleanliness**: Every modification must be surgically precise—incorrect memory writes can crash the process (cause organ failure)
- **Post-operative care**: The surgical team must ensure the modifications integrate properly (instruction cache flushing, signal handling)

Interface and Responsibilities

The Runtime Patcher provides a clean API for attaching to processes, reading/writing their memory, and modifying code pages. This interface isolates the complexities of `ptrace` and system call injection from the rest of the instrumentation pipeline.

Method Name	Parameters	Returns	Description
`attach_to_process`	`pid pid_t`	`int` (0 for success, error code otherwise)	Attaches to the target process using `PTRACE_ATTACH`, stops it, and prepares for memory operations. Returns error if process doesn't exist or permission is denied.
`detach_from_process`	`pid pid_t`	`int`	Detaches from the process using `PTRACE_DETACH`, allowing it to resume normal execution. Must clean up any temporary modifications to process state.
`read_process_memory`	`pid pid_t`, `addr uint64_t`, `buffer void*`, `size size_t`	`int`	Reads `size` bytes from the process's virtual address `addr` into local `buffer`. Uses `PTRACE_PEEKDATA` in word-sized chunks. Returns bytes read or error.
`write_process_memory`	`pid pid_t`, `addr uint64_t`, `buffer void*`, `size size_t`	`int`	Writes `size` bytes from `buffer` into the process's virtual address `addr`. Uses `PTRACE_POKEDATA` word-by-word. Returns bytes written or error.
`make_page_writable`	`pid pid_t`, `addr uint64_t`, `size size_t`	`int`	Changes memory protection of the page(s) containing `[addr, addr+size)` to include write permission. Injects an `mprotect` syscall into the process via `ptrace`. Returns 0 on success.
`flush_instruction_cache`	`pid pid_t`, `addr uint64_t`, `size size_t`	`int`	Ensures the CPU instruction cache is synchronized after code modifications. On x86-64, may be a no-op; on ARM, uses `__builtin___clear_cache` or similar.
`inject_trampoline_runtime`	`pid pid_t`, `target_addr uint64_t`, `trampoline trampoline_t*`	`int`	Complete runtime hooking workflow: makes page writable, writes trampoline at `target_addr`, flushes cache. Returns success/failure.

Responsibilities Summary:

1. **Process Lifecycle Management**: Safely attach to and detach from running processes without causing crashes or hangs
2. **Memory Access**: Read and write process memory with proper error handling and atomicity guarantees
3. **Memory Protection Manipulation**: Temporarily modify page permissions to allow code modification
4. **Cache Coherence**: Ensure CPU caches reflect memory modifications
5. **State Preservation**: Maintain and restore process state (registers, signals) during instrumentation

Internal Behavior: Live Patching Protocol

The runtime patcher follows a strict protocol to minimize risk and ensure reliable modifications. The algorithm proceeds in these numbered steps:

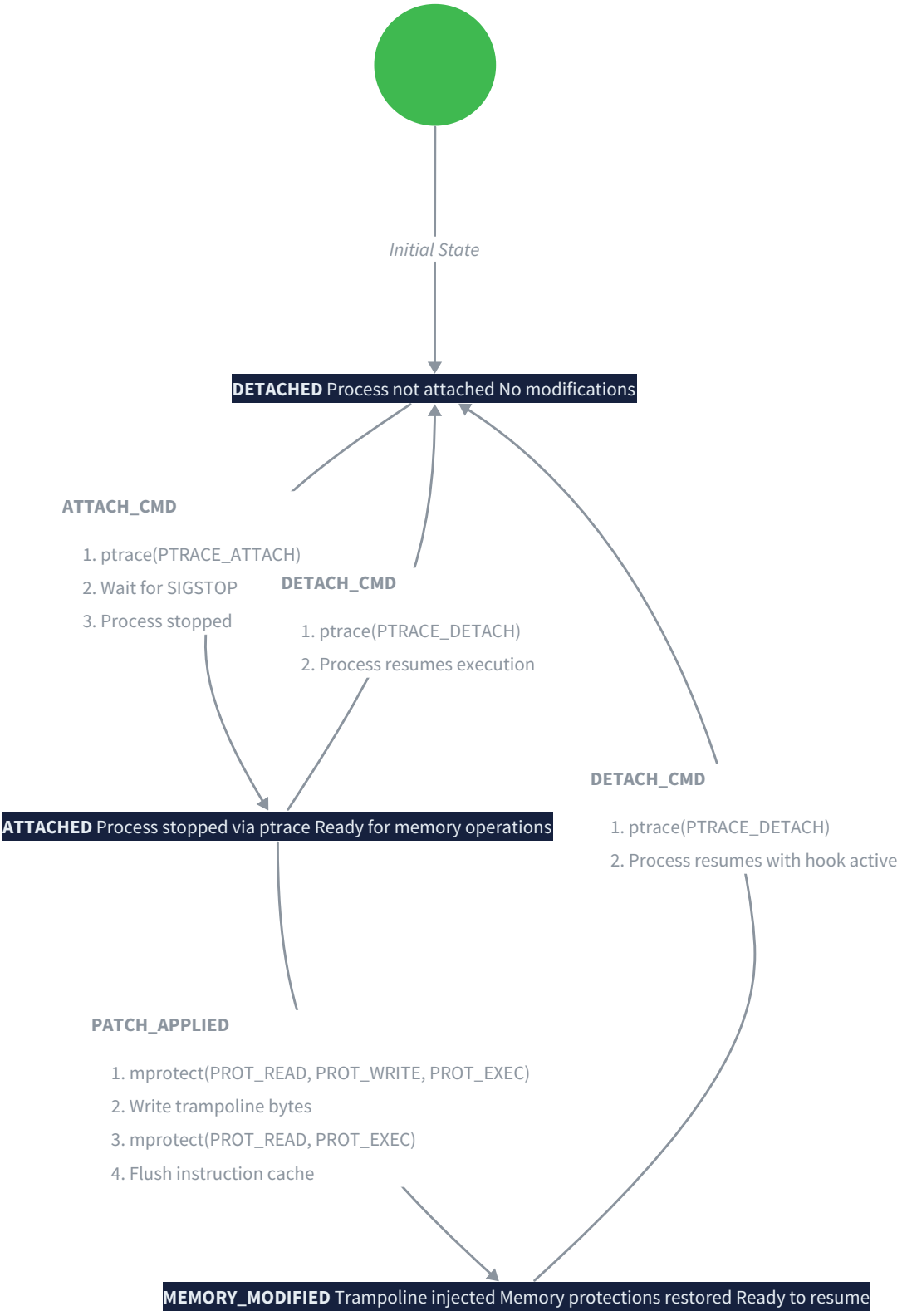
1. **Process Attachment and Stopping**
   - Call `attach_to_process(pid)` which internally uses `ptrace(PTRACE_ATTACH, pid, NULL, NULL)`
   - Wait for the process to stop using `waitpid(pid, &status, 0)`
   - Verify the process stopped due to `SIGSTOP` (status should be `0x137F` for `WIFSTOPPED` and `WSTOPSIG(status) == SIGSTOP`)
2. **Memory Protection Modification**
   - Calculate the page boundaries containing the target address: `page_start = target_addr & ~(PAGE_SIZE-1)`
   - Call `make_page_writable(pid, page_start, PAGE_SIZE)` which: a. Reads the current process registers with `ptrace(PTRACE_GETREGS, pid, NULL, &regs)` b. Saves original register state (especially `%rip`, `%rax`, `%rdi`, `%rsi`, `%rdx`) c. Injects an `mprotect` syscall by setting `%rax` to `10` (syscall number for `mprotect`), `%rdi` to `page_start`, `%rsi` to `PAGE_SIZE`, `%rdx` to `PROT_READ|PROT_WRITE|PROT_EXEC` d. Executes the syscall with `ptrace(PTRACE_SYSCALL, pid, NULL, NULL)` and waits e. Restores original registers after syscall completion
3. **Trampoline Injection**
   - Use `write_process_memory(pid, target_addr, trampoline->code, trampoline->code_size)` to write the trampoline bytes
   - This writes word-by-word (8 bytes at a time on x86-64) using `PTRACE_POKEDATA`
   - Verify writes by reading back and comparing with original bytes
4. **Cache Flushing**
   - Call `flush_instruction_cache(pid, target_addr, trampoline->code_size)`
   - On x86-64, this may involve a serializing instruction like `cpuid` or simply returning success (x86 has coherent caches)
   - On ARM architectures, must use explicit cache flush via `__builtin___clear_cache` or similar
5. **Memory Protection Restoration** (Optional but recommended)
   - Restore original page permissions using another injected `mprotect` syscall with original flags
   - This prevents accidental corruption and maintains security invariants
6. **Process Resumption**
   - Call `detach_from_process(pid)` which internally uses `ptrace(PTRACE_DETACH, pid, NULL, NULL)`
   - The process resumes execution with the new trampoline active

Concrete Example Walkthrough: Consider hooking the `printf` function in a running `nginx` process (PID 1234). The function resides at virtual address `0x555555554a30`. The protocol executes as:

1. Attach to PID 1234, process stops with `SIGSTOP`
2. Page containing `0x555555554a30` is `0x555555554000` (page-aligned). Inject `mprotect(0x555555554000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC)`
3. Write 14-byte trampoline (`push %rax; movabs $0x7ffff7abcde0, %rax; call *%rax; pop %rax; jmp 0x555555554a3a`) at `0x555555554a30`
4. Flush instruction cache for address range `0x555555554a30-0x555555554a3e`
5. Restore original page permissions (`PROT_READ|PROT_EXEC`)
6. Detach from process; `nginx` continues with `printf` calls now routed through our hook

State Machine: Ptrace Session Lifecycle

The Runtime Patcher maintains an explicit state machine to ensure operations occur in the correct order and to prevent illegal transitions (like writing memory before attaching). The state machine defines the lifecycle of a patching session.


Initial State
DETACHED Process not attached No modifications
ATTACH_CMD
1. ptrace(PTRACE_ATTACH)
2. Wait for SIGSTOP
3. Process stopped
DETACH_CMD
1. ptrace(PTRACE_DETACH)
2. Process resumes execution
ATTACHED Process stopped via ptrace Ready for memory operations
DETACH_CMD
1. ptrace(PTRACE_DETACH)
2. Process resumes with hook active
PATCH_APPLIED
1. mprotect(PROT_READ, PROT_WRITE, PROT_EXEC)
2. Write trampoline bytes
3. mprotect(PROT_READ, PROT_EXEC)
4. Flush instruction cache
MEMORY_MODIFIED Trampoline injected Memory protections restored Ready to resume

Current State	Event	Next State	Actions Taken
`DETACHED`	`ATTACH_CMD` (call `attach_to_process`)	`ATTACHED`	Send `PTRACE_ATTACH`, wait for `SIGSTOP`, verify process stopped
`ATTACHED`	`DETACH_CMD` (call `detach_from_process`)	`DETACHED`	Send `PTRACE_DETACH`, process resumes execution
`ATTACHED`	`MODIFY_MEMORY` (begin memory write)	`MEMORY_MODIFIED`	Change page protections via injected `mprotect`, begin writing memory
`MEMORY_MODIFIED`	`WRITE_COMPLETE` (memory write finished)	`ATTACHED`	Flush instruction cache, optionally restore page protections
`ATTACHED`	`SIGNAL_RECEIVED` (process received signal)	`ATTACHED`	Intercept signal via `PTRACE_GETSIGINFO`, optionally suppress or forward it
Any state	`ERROR_DETECTED` (ptrace fails, process died)	`ERROR`	Log error, attempt cleanup, transition to terminal error state
`ERROR`	`CLEANUP`	`DETACHED`	Force detach if still attached, free resources, reset state

State Descriptions:

- **`DETACHED`**: Initial and final state. No connection to target process. Can only transition to `ATTACHED`.
- **`ATTACHED`**: Process is stopped via `ptrace`. Memory can be read/written, but code pages are still protected. Signals are queued.
- **`MEMORY_MODIFIED`**: Page protections have been changed and memory is being written. Must complete write and flush cache before detaching.
- **`ERROR`**: Terminal error state. Requires cleanup before returning to `DETACHED`.

Signal Handling Consideration: When in `ATTACHED` state, the patcher must handle pending signals. The proper approach is to use `PTRACE_GETSIGINFO` to inspect the signal, then either suppress it (if it's `SIGSTOP` from our attach) or forward it after detaching. For other signals, we can use `PTRACE_CONT` with the signal number to deliver it, then wait for the process to stop again.

ADR: Ptrace vs. /proc/pid/mem for Memory Access

> **Decision: Use ptrace for memory access and control**
>
> - **Context**: We need to read and write memory in a running process for runtime instrumentation. Two primary Linux interfaces exist: the `ptrace` system call (used by debuggers like GDB) and the `/proc/pid/mem` virtual file (memory access via file operations).
> - **Options Considered**:
>   1. **`ptrace` with `PTRACE_PEEKDATA` / `PTRACE_POKEDATA`**: Traditional debugger interface providing atomic word-sized memory operations
>   2. **`/proc/pid/mem` with `pread` / `pwrite`**: Memory-mapped file interface allowing arbitrary-sized reads/writes
>   3. **Process VM via `process_vm_readv` / `process_vm_writev`**: System calls for fast cross-process memory copying
> - **Decision**: Use `ptrace` for all memory access and process control.
> - **Rationale**:
>   - **Atomicity and consistency**: `ptrace` operations are guaranteed atomic at the word level and properly synchronized with the traced process's execution state
>   - **Single interface for all operations**: `ptrace` provides attachment/detachment, register access, syscall injection, and signal handling—not just memory access
>   - **Wider compatibility**: `/proc/pid/mem` requires the process to be stopped (via `ptrace` anyway), and some systems restrict `/proc` access
>   - **Security model alignment**: `ptrace` follows the standard Unix debugging permission model (`ptrace_scope` , YAMA), which is well-understood
> - **Consequences**:
>   - **Performance overhead**: `ptrace` requires a context switch per memory word (8 bytes on x86-64), making bulk operations slower
>   - **Complexity**: Must handle `ptrace` quirks like `EIO` errors on unmapped addresses, signal interference
>   - **Portability**: Linux-specific (though our tool targets Linux ELF binaries anyway)

Option	Pros	Cons	Why Not Chosen?
`ptrace`	Atomic operations, integrated process control, universal Linux support, proper signal handling	Slow for bulk transfers, complex error handling, per-word system calls	**CHOSEN** - Provides complete control needed for safe instrumentation
`/proc/pid/mem`	Fast arbitrary-sized reads/writes, simpler API for bulk memory access	Requires process to be stopped (needs `ptrace` anyway), permissions issues, no register/syscall access	Requires `ptrace` anyway for stopping process; adds complexity of two interfaces
`process_vm_readv/writev`	Very fast, designed for cross-process memory, arbitrary sizes	Requires process to be stopped for code modification, no process control capabilities	Insufficient for our needs—can't change page protections or inject syscalls

Common Pitfalls

⚠️ Pitfall: Forgetting to handle pending signals during ptrace

- **Description**: After attaching with `PTRACE_ATTACH`, the process receives `SIGSTOP`. However, other signals may already be pending or arrive while the process is stopped. If not handled, they can be lost or cause unexpected behavior when the process resumes.
- **Why it's wrong**: Unhandled signals can cause the process to terminate unexpectedly (e.g., `SIGTERM`) or behave incorrectly (signals are part of the process's normal operation).
- **How to fix**: After attaching and waiting for the stop, check for other signals using `PTRACE_GETSIGINFO`. If signals other than `SIGSTOP` are pending, either suppress them (if they're from the instrumentation) or deliver them with `PTRACE_CONT` before making modifications. Always use `PTRACE_SYSCALL` or `PTRACE_SYSEMU` when injecting syscalls to properly handle signal interrupts.

⚠️ Pitfall: Not flushing the instruction cache after code modification

- **Description**: On some architectures (ARM, PowerPC, MIPS), the CPU instruction cache is not coherent with data cache. Modified code in memory may not be visible to the CPU until the instruction cache is explicitly flushed.
- **Why it's wrong**: The CPU continues executing old, cached instructions instead of the newly written trampoline, causing the hook to not trigger.
- **How to fix**: After writing trampoline code, call `__builtin___clear_cache((void*)addr, (void*)(addr + size))` (GCC/Clang intrinsic) or use the `cacheflush` syscall on Linux. On x86-64, this is largely unnecessary (caches are coherent), but still good practice for portability.

⚠️ **Pitfall: Assuming code pages are writable by default**

- **Description**: Executable pages in modern systems are typically mapped `PROT_READ|PROT_EXEC` (no write permission) to prevent code injection attacks. Direct `PTRACE_POKEDATA` to such pages will fail with `EIO`.
- **Why it's wrong**: The memory write fails silently or with an error, leaving the trampoline unwritten while the tool thinks it succeeded.
- **How to fix**: Always call `make_page_writable` before writing to code pages. This function must inject an `mprotect` syscall into the target process to temporarily add `PROT_WRITE` permission. Remember to restore original permissions after writing.

⚠️ **Pitfall: Modifying multi-threaded processes without stopping all threads**

- **Description**: When attaching to a multi-threaded process, `PTRACE_ATTACH` only attaches to the specific thread (process) with the given PID. Other threads continue running and may execute code being modified.
- **Why it's wrong**: Race condition: another thread might execute the code being patched mid-modification, causing crashes or undefined behavior.
- **How to fix**: For production instrumentation, consider using `PTRACE_SEIZE` with `PTRACE_INTERRUPT` to stop all threads, or iterate through all thread IDs in `/proc/[pid]/task/` and attach to each. For simplicity in Milestone 4, we assume single-threaded processes or accept the risk for educational purposes.

⚠️ **Pitfall: Not preserving register state during syscall injection**

- **Description**: When injecting an `mprotect` syscall via `ptrace`, the patcher overwrites the process's registers (especially `%rip`, `%rax`, `%rdi`, etc.). If original values aren't saved and restored, the process resumes execution with corrupted state.
- **Why it's wrong**: The process crashes immediately after resuming because its instruction pointer or other critical registers point to invalid addresses.
- **How to fix**: Before modifying registers, use `PTRACE_GETREGS` to save all registers. After the syscall completes, use `PTRACE_SETREGS` to restore them exactly. Be careful with syscall return values—store them in a scratch register or stack location if needed.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Process Attachment	Basic `ptrace(PTRACE_ATTACH)` with error checking	`PTRACE_SEIZE` with `PTRACE_INTERRUPT` for cleaner attachment
Memory Access	Word-by-word `PTRACE_PEEKDATA` / `PTRACE_POKEDATA` loops	Hybrid: bulk reads via `/proc/pid/mem`, writes via `ptrace` for atomicity
Syscall Injection	Direct register manipulation with `PTRACE_SYSCALL`	`PTRACE_SYSEMU` for precise syscall emulation control
Cache Flushing	`__builtin___clear_cache` intrinsic	Architecture-specific: `cacheflush` syscall for ARM, `cpuid` for x86

Recommended File/Module Structure:

```
elflex/
├── src/
│   ├── main.c                      # CLI entry point
│   ├── elfloader/                  # ELF parsing (Milestone 1)
│   ├── patcher/                    # Static patching (Milestone 1)
│   ├── trampoline/                 # Trampoline generation (Milestone 2)
│   ├── injector/                   # Code injection (Milestone 3)
│   ├── runtime/                    # Runtime patching (Milestone 4) ← THIS COMPONENT
│   │   ├── runtime.c               # Main runtime patcher implementation
│   │   ├── runtime.h               # Public interface declarations
│   │   ├── ptrace_helpers.c        # Low-level ptrace wrappers
│   │   └── syscall_inject.c        # Syscall injection utilities
│   └── utils/
│       ├── error.c                 # Error handling
│       └── logging.c               # Debug logging
└── tests/
    └── test_runtime.c              # Runtime patching tests
```

Infrastructure Starter Code (Complete ptrace helpers):

C

```c
/* ptrace_helpers.h */

#ifndef PTRACE_HELPERS_H

#define PTRACE_HELPERS_H


#include <sys/types.h>

#include <sys/ptrace.h>

#include <sys/wait.h>

#include <stdint.h>


/* Wait for process to stop after ptrace operation */

int wait_for_stop(pid_t pid, int *status);


/* Read a word from process memory at address addr */

uint64_t ptrace_peekdata(pid_t pid, uint64_t addr, int *error);


/* Write a word to process memory at address addr */

int ptrace_pokedata(pid_t pid, uint64_t addr, uint64_t data);


#endif /* PTRACE_HELPERS_H */
```

```c
/* ptrace_helpers.c */

#include "ptrace_helpers.h"

#include <errno.h>

#include <stdio.h>


int wait_for_stop(pid_t pid, int *status) {

    if (waitpid(pid, status, 0) == -1) {

        perror("waitpid failed");

        return -1;

    }



    if (!WIFSTOPPED(*status)) {

        fprintf(stderr, "Process did not stop (status: 0x%x)\n", *status);

        return -1;

    }



    return 0;

}


uint64_t ptrace_peekdata(pid_t pid, uint64_t addr, int *error) {

    errno = 0;

    long result = ptrace(PTRACE_PEEKDATA, pid, (void*)addr, NULL);



    if (errno != 0) {

        if (error) *error = errno;

        return 0;

    }



    if (error) *error = 0;

    return (uint64_t)result;

}


int ptrace_pokedata(pid_t pid, uint64_t addr, uint64_t data) {

    errno = 0;

    ptrace(PTRACE_POKEDATA, pid, (void*)addr, (void*)data);



    return (errno == 0) ? 0 : -1;
```

```
}
```

Core Logic Skeleton Code (Main runtime patching functions with TODOs):

```c
/* runtime.c - Main runtime patcher implementation */

#include "runtime.h"

#include "ptrace_helpers.h"

#include <sys/uio.h>

#include <sys/mman.h>

#include <errno.h>

#include <string.h>


/* State tracking for runtime patching session */

typedef struct {

    pid_t pid;

    int attached;

    uint64_t original_regs[16];  /* Scratch for register saving */

    /* Add other state as needed */

} runtime_session_t;


int attach_to_process(pid_t pid) {

    // TODO 1: Check if pid exists (kill(pid, 0) can check)

    // TODO 2: Call ptrace(PTRACE_ATTACH, pid, NULL, NULL)

    // TODO 3: If successful, wait for process to stop using wait_for_stop

    // TODO 4: Verify stop reason is SIGSTOP (WSTOPSIG(status) == SIGSTOP)

    // TODO 5: Set session state to ATTACHED

    // TODO 6: Return 0 on success, error code on failure

    return -1; /* Placeholder */

}


int detach_from_process(pid_t pid) {

    // TODO 1: Verify we're attached to this pid (session state)

    // TODO 2: Call ptrace(PTRACE_DETACH, pid, NULL, NULL)

    // TODO 3: If detach fails with ESRCH, process may have died (log warning)

    // TODO 4: Clear session state to DETACHED

    // TODO 5: Return 0 on success, error code otherwise

    return -1; /* Placeholder */

}


int read_process_memory(pid_t pid, uint64_t addr, void *buffer, size_t size) {

    // TODO 1: Validate parameters (addr alignment, buffer not NULL, size > 0)
```

```c
    // TODO 2: For each 8-byte chunk in [addr, addr+size):
    //   a. Call ptrace_peekdata(pid, current_addr, &error)
    //   b. If error != 0, break and return partial read count
    //   c. Copy up to min(remaining, 8) bytes to buffer
    //   d. Advance buffer pointer and address
    // TODO 3: Return total bytes successfully read
    return -1; /* Placeholder */
}

int write_process_memory(pid_t pid, uint64_t addr, const void *buffer, size_t size) {
    // TODO 1: Validate parameters
    // TODO 2: For each 8-byte chunk in [addr, addr+size):
    //   a. Build a 64-bit word: existing bytes from process OR new bytes from buffer
    //   b. Call ptrace_pokedata(pid, current_addr, word)
    //   c. If failed, break and return bytes written so far
    //   d. Verify write by reading back and comparing (optional but recommended)
    //   e. Advance buffer pointer and address
    // TODO 3: Return total bytes successfully written
    return -1; /* Placeholder */
}

int make_page_writable(pid_t pid, uint64_t addr, size_t size) {
    // TODO 1: Calculate page-aligned address range: page_start = addr & ~(PAGE_SIZE-1)
    // TODO 2: Read current process registers with ptrace(PTRACE_GETREGS, ...)
    // TODO 3: Save original register values (especially RAX, RDI, RSI, RDX, RIP)
    // TODO 4: Set up mprotect syscall:
    //   - RAX = 10 (syscall number for mprotect on x86-64)
    //   - RDI = page_start
    //   - RSI = size rounded up to page boundary
    //   - RDX = PROT_READ | PROT_WRITE | PROT_EXEC (0x7)
    // TODO 5: Write modified registers back with ptrace(PTRACE_SETREGS, ...)
    // TODO 6: Single-step or continue to syscall: ptrace(PTRACE_SYSCALL, ...)
    // TODO 7: Wait for syscall stop, check return value in RAX (0 for success)
    // TODO 8: Restore original registers
    // TODO 9: Return 0 if mprotect succeeded, error code otherwise
    return -1; /* Placeholder */
```

```c
}

int flush_instruction_cache(pid_t pid, uint64_t addr, size_t size) {
    // TODO 1: On x86-64, instruction cache is coherent - return success (0)
    // TODO 2: On ARM/MIPS/PowerPC, use __builtin___clear_cache
    //   Note: Must execute in target process context - consider injecting
    //   a syscall to cacheflush or using PTRACE_CALL to run clear_cache
    // TODO 3: For simplicity in Milestone 4, we can assume x86-64 and return 0
    return 0; /* x86-64 cache is coherent */
}

int inject_trampoline_runtime(pid_t pid, uint64_t target_addr, trampoline_t *trampoline) {
    // TODO 1: Call attach_to_process if not already attached
    // TODO 2: Call make_page_writable for the page containing target_addr
    // TODO 3: Call write_process_memory to write trampoline->code at target_addr
    // TODO 4: Call flush_instruction_cache for the written region
    // TODO 5: Optionally restore original page permissions (another mprotect)
    // TODO 6: Return success (0) if all steps succeeded
    return -1; /* Placeholder */
}
```

Language-Specific Hints (C):

- **Error Handling**: Always check `errno` after `ptrace` calls. Common errors: `ESRCH` (process doesn't exist), `EIO` (invalid address), `EPERM` (insufficient permissions).
- **Register Access**: Use `struct user_regs_struct` from `<sys/user.h>` for portable register access with `PTRACE_GETREGS` / `PTRACE_SETREGS`.
- **Signal Handling**: Use `struct sigaction` and `PTRACE_GETSIGINFO` / `PTRACE_SETSIGINFO` for fine-grained signal control.
- **Memory Alignment**: x86-64 allows unaligned accesses, but `ptrace` operations are most efficient on word-aligned addresses. Round addresses down to 8-byte boundaries for reads/writes.
- **System Call Numbers**: Find architecture-specific syscall numbers in `/usr/include/asm/unistd_64.h`. `mprotect` is 10 on x86-64, `cacheflush` is not a standard Linux syscall (use `__builtin___clear_cache`).

Milestone Checkpoint for Runtime Patching:

To verify successful implementation of the Runtime Patcher:

1. **Test with a simple target process**:

```bash
# Compile a test program that runs in a loop
cat > test_loop.c << 'EOF'
#include <stdio.h>
#include <unistd.h>

void target_function() {
    printf("Target function called\n");
}

int main() {
    while (1) {
        target_function();
        sleep(2);
    }
    return 0;
}
EOF
gcc -o test_loop test_loop.c
./test_loop &
TARGET_PID=$!
```

2. **Run your runtime patcher to hook `target_function`**:

```bash
./elflex --runtime --pid $TARGET_PID --hook target_function --hook-lib ./my_hook.so
```

3. **Expected behavior**: The test process continues running but now prints additional output from your hook function before each "Target function called" line.

4. **Verification commands**:

```bash
# Check if process is still running
ps -p $TARGET_PID

# Look for trampoline code in process memory
gdb -p $TARGET_PID -ex "disas target_function" -ex "quit"
# Should show a jump instruction at the start of target_function

# Check logs (if your tool logs to file)
tail -f /tmp/elflex.log  # If you implement logging
```

5. **Signs of success**: Process continues running indefinitely with hook output. Process does not crash or hang.
6. **Signs of failure**: Process crashes, hangs, or hook is not called. Use `strace -p $TARGET_PID` to see system calls, or `gdb` to examine memory and registers.

Debugging Tips Table:

Symptom	Likely Cause	How to Diagnose	Fix
Process crashes immediately after patching	Register corruption during syscall injection	Use `gdb` to attach after crash, examine register values at crash point	Ensure all registers are saved and restored exactly during `make_page_writable`
Hook not called; original function executes normally	Trampoline not written or instruction cache not flushed	Read back memory at target address: `gdb -p $PID -ex "x/10i 0xADDR"`	Verify `write_process_memory` succeeds; add explicit cache flush even on x86
`ptrace` returns `EPERM` even as root	YAMA ptrace_scope restrictions	Check `cat /proc/sys/kernel/yama/ptrace_scope` (1 or 2 restricts)	Set `ptrace_scope` to 0 temporarily: `echo 0 > /proc/sys/kernel/yama/ptrace_scope`
Process hangs after `PTRACE_ATTACH`	Signal not delivered or waitpid timing issue	Use `strace` on your patcher tool to see which syscall hangs	Ensure proper `waitpid` after attach; check for zombie processes
Can read memory but not write	Page permissions not changed	Check `/proc/$PID/maps` to see page flags	Ensure `make_page_writable` injects `mprotect` correctly; verify syscall return value
Multi-threaded process behaves oddly after patching	Other threads executing while main thread stopped	Check `/proc/$PID/task/` for other threads	Consider attaching to all threads or accepting this limitation for Milestone 4

Interactions and Data Flow

> **Milestone(s):** 1 (Instruction-level Patching), 2 (Function Hooking with Trampolines), 3 (Code Injection), 4 (Runtime Patching via ptrace)

The **Interactions and Data Flow** section describes how the architectural components collaborate to achieve instrumentation goals. While previous sections examined components in isolation, this section reveals the orchestrated workflow—showing how data moves through the system, how components hand off responsibilities, and how the overall architecture supports both static and dynamic instrumentation scenarios.

Understanding these flows is critical because binary instrumentation is fundamentally a **pipeline of transformations**: raw binary → parsed representation → modified representation → final output (either patched file or running process). Each stage builds upon the previous, and failures at any point must be handled gracefully to avoid corrupting the target.

Static Instrumentation Flow

Mental Model: The Assembly Line Think of static instrumentation as an assembly line in a factory. The raw material (original ELF binary) enters at one end, moves through specialized workstations (components), and emerges as a modified product (patched binary). Each workstation performs a specific, isolated transformation, with quality checks ensuring the product remains valid throughout. The line can be configured for different jobs: simple instruction replacement (Milestone 1) or complex function hooking with code injection (Milestones 2-3).

The static flow encompasses all operations performed on a binary file while it's not executing. This includes the core functionality from Milestones 1-3: basic instruction patching, function hooking via trampolines, and injection of new code sections. The flow follows a **load** → **analyze** → **transform** → **write** pattern, with optional cycles for multiple patches.

End-to-End Workflow

The complete static instrumentation process involves seven coordinated steps, each managed by a specific component or set of components. The following diagram illustrates this sequence:

Step-by-Step Algorithm:

1. **Binary Loading and Parsing** (ELF Loader)
   - The user invokes the tool with the target binary path and instrumentation specification (e.g., "hook function `printf` with hook `my_printf`").
   - The tool calls `elf_load()` which reads the entire file into memory, validates ELF headers, and parses section/segment headers.
   - The ELF Loader constructs an `elf_binary_t` structure containing:
     - The original file buffer (preserved for reference)
     - Parsed section headers mapped to `section_t` objects
     - Parsed program headers mapped to `segment_t` objects
     - Direct pointers to key sections (`.text` , `.symtab` , `.strtab`)
   - If loading fails (invalid ELF, permissions), the tool reports the error and exits cleanly.
2. **Target Identification and Planning** (Patch Planner + ELF Loader)
   - For simple instruction patching: The tool extracts the target offset from user input (hexadecimal or symbol name). If a symbol name is provided, `elf_get_symbol_address()` resolves it to a virtual address, which is then converted to a file offset via section mapping.
   - For function hooking: The tool resolves the target function symbol to its entry point address using the symbol table. It also resolves the hook function symbol (if already present in the binary) or prepares to inject it.
   - The Patch Planner calls `plan_patch()` to analyze the target location:
     - Validates the offset lies within an executable section (typically `.text`)
     - For instruction replacement, ensures the new instruction sequence fits (or plans NOP padding)
     - For function hooking, determines the size of the prologue to replace (typically 5-14 bytes for a jump)
     - Creates a `patch_site_t` structure with all planning metadata
3. **Trampoline Generation** (Trampoline Generator, optional for Milestone 2+)
   - If the instrumentation involves function hooking, the tool calls `generate_trampoline()` :
     - Determines the hook function address (either existing symbol or planned injection address)
     - Extracts the original function's prologue instructions at the entry point
     - Calls `relocate_prologue()` to create a movable version of these instructions

        - Applies `fixup_rip_relative()` to adjust any RIP-relative addressing in relocated instructions
        - Generates trampoline assembly: register save → call hook → register restore → relocated prologue → jump to original function body
        - Encodes the trampoline into position-independent machine code
    - The output is a `trampoline_t` structure containing the code and metadata.
4. **Code Injection Preparation** (Code Injector, optional for Milestone 3)
    - If the hook function doesn't exist in the original binary, the tool prepares to inject it:
        - Compiles the hook function (supplied as C source) to position-independent machine code (external step)
        - Calls `inject_section()` with the compiled code, which:
            - Finds suitable space (via `allocate_code_cave()` or segment expansion)
            - Creates a new `.inject` section header with `SHF_ALLOC | SHF_EXECINSTR` flags
            - Updates program headers if a new `PT_LOAD` segment is needed
            - Applies relocations to resolve references to symbols in the original binary (`resolve_symbol_in_target()`)
        - Returns the virtual address where the hook will reside
    - For trampolines that will be placed in the injected section (rather than inline), the trampoline code is added to the injection payload.
5. **Patch Application** (Patch Planner + Binary Patcher)
    - The tool calls `apply_patch()` for each planned modification:
        - For simple instruction replacement: Overwrites bytes at the target offset in the `.text` section's data buffer, using NOP padding if `new_size < original_size` .
        - For function hooking: Replaces the function prologue with a jump to the trampoline. The jump instruction is encoded as either:
            - Relative jump (`E9`) if the trampoline is within ±2GB
            - Indirect jump via register (`FF 25`) if the trampoline is distant (e.g., in injected section)
        - Updates the `patch_site_t.applied` flag to `1` .
    - If multiple patches overlap or conflict, the tool detects this and reports `ERR_PATCH_OVERLAP` .
6. **Binary Reconstruction** (implicit Binary Writer)
    - After all patches are applied, the tool must write the modified `elf_binary_t` back to disk:
        - Updates section header offsets if any sections grew or were added
        - Recalculates segment file offsets and memory sizes if segments changed
        - Updates the ELF header's `e_shoff` and `e_phnum` if sections/segments were added
        - Writes all headers, then section/segment data, to a new file
    - The writer ensures proper alignment (16-byte for sections, page-size for segments) and maintains structural validity.
7. **Cleanup and Validation**
    - The tool frees all allocated resources: `elf_binary_free()` , `patch_free()` , `trampoline_free()` , etc.
    - Optionally, the tool validates the output binary using `readelf` or a simple execution test to ensure it loads and runs.

Data Flow Through Components:

Component	Input	Processing	Output
ELF Loader	File path string	Parse headers, load sections	`elf_binary_t*` with complete representation
Patch Planner	Target offset/symbol, new bytes	Validate, calculate padding, plan patch	`patch_site_t*` with patch metadata
Trampoline Generator	Target function address, hook address	Generate trampoline code, relocate prologue	`trampoline_t*` with executable trampoline
Code Injector	Hook code buffer, size	Allocate space, create section, resolve symbols	Virtual address of injected code, updated `elf_binary_t`
Binary Patcher	`elf_binary_t*`, `patch_site_t*`	Modify section data buffers	Updated `elf_binary_t*` with applied patches
Binary Writer	`elf_binary_t*`, output path	Reconstruct headers, write file	New ELF file on disk

Concrete Example: Hooking `malloc`

Consider instrumenting `libc.so` to log all memory allocations:

1. **Load**: `elf_load("libc.so.6")` → creates `elf_binary_t` with 200+ sections.
2. **Plan**: `elf_get_symbol_address(binary, "malloc")` → returns `0xabc00`. `plan_patch(binary, 0xabc00, NULL, 0)` → analyzes prologue at that address.
3. **Generate**: `generate_trampoline(0xabc00, HOOK_ADDR)` → creates trampoline that calls `log_malloc` before original `malloc`.
4. **Inject**: `inject_section(binary, ".inject", log_malloc_code, size, SHF_ALLOC|SHF_EXECINSTR)` → adds new section at `0x120000`.
5. **Apply**: `apply_patch(binary, patch)` → replaces first 5 bytes of `malloc` with `E9 fb 54 01 00` (jump to trampoline at `0x120000`).
6. **Write**: `binary_write(binary, "libc_instrumented.so")` → produces modified library.
7. **Test**: `LD_PRELOAD=./libc_instrumented.so myapp` → allocations are logged.

Architecture Decision: Sequential vs. Parallel Patch Application

> **Decision: Sequential, Ordered Patch Application**
>
> - **Context**: When multiple hooks or patches are requested in a single run, they may have dependencies (e.g., hook A calls original function B, which is also being hooked). The system must apply patches in a way that maintains correctness.
> - **Options Considered**:
>   1. **Apply all patches simultaneously**: Calculate all final byte values and write them in one pass. This assumes patches don't interfere.
>   2. **Apply sequentially in user order**: Process patches as they appear in the command line or configuration file.
>   3. **Apply sequentially with dependency ordering**: Analyze symbol dependencies and apply patches in topological order.
> - **Decision**: Option 2 — sequential application in user-specified order.
> - **Rationale**: The tool's primary use case is hooking independent functions or making isolated patches. Users can reason about order if dependencies exist. Implementing dependency analysis adds significant complexity (full call graph analysis) for limited benefit, especially since the trampoline design already handles most dependency cases (the hook calls the original function via the trampoline, which remains valid even if the original is patched).
> - **Consequences**: Users must specify patches in correct order if dependencies exist. The implementation remains simpler and faster. Potential for misuse if users hook a function and its hook recursively without care.

Option	Pros	Cons	Chosen?
Simultaneous application	Fast, atomic	Cannot handle patches that affect same bytes interdependently	No
Sequential in user order	Simple, predictable	User must understand dependencies	**Yes**
Dependency ordering	Handles complex cases automatically	Requires full program analysis, complex implementation	No

Common Pitfalls in Static Flow

⚠️ Pitfall: Offset Miscalculation Between Virtual and File Addresses

- **Description**: Using a virtual address (from `elf_get_symbol_address()`) directly as a file offset when patching, causing corruption of wrong bytes.
- **Why it's wrong**: Virtual addresses (VMA) map to memory at runtime; file offsets map to locations in the ELF file. They differ by the section's load offset (`sh_offset` vs `sh_addr`). The `.text` section might be at file offset `0x400` but load at virtual address `0x400000`.
- **Fix**: Always convert virtual addresses to file offsets using `file_offset = vaddr - section->sh_addr + section->sh_offset`. The `plan_patch()` function should handle this internally.

⚠️ Pitfall: Forgetting to Update All Affected Headers After Injection

- **Description**: Adding a new section but only updating the section header table, leaving program headers pointing to old offsets, causing loader failures.
- **Why it's wrong**: ELF loaders use program headers (`PT_LOAD` segments) to map file regions into memory. If a new section extends a segment, the segment's `p_filesz` and `p_memsz` must increase. If a new segment is added, the program header table must be expanded and `e_phnum` updated.
- **Fix**: After any injection, recalculate all segment boundaries and update both section and program headers. Use a helper function that scans all sections belonging to each segment.

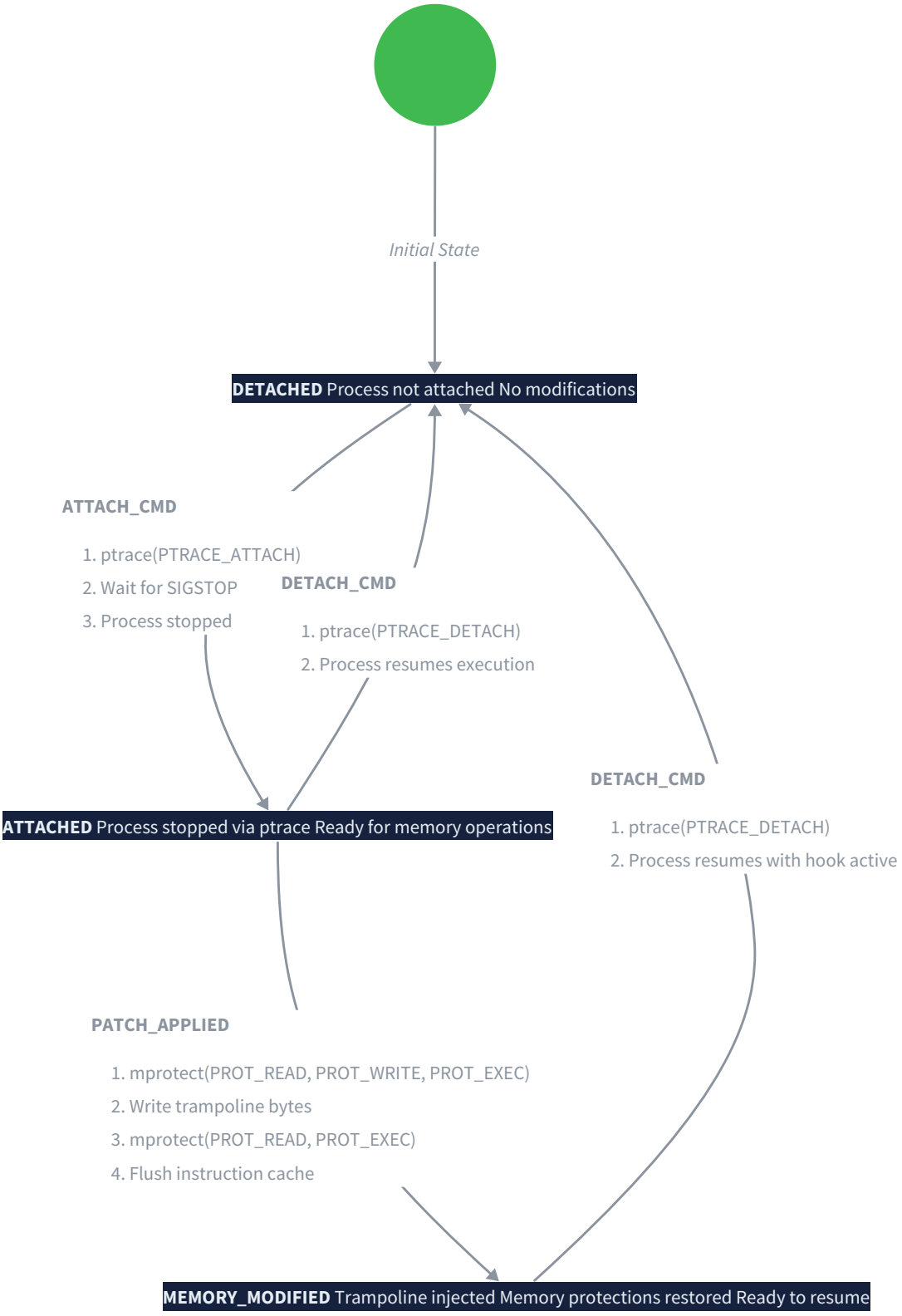
Runtime Instrumentation Flow

Mental Model: The Live Surgery Team Runtime instrumentation is like a surgical team performing a procedure on a living patient. The patient (target process) must be kept alive but temporarily paused. The surgeons (tool components) use specialized instruments (ptrace) to access internal organs (memory), make precise modifications (inject trampolines), then close up and let the patient resume normal life—now with enhanced functionality (hooks). The challenge is minimizing "anesthesia time" (process stoppage) and avoiding complications (crashes, memory corruption).

The runtime flow implements Milestone 4, enabling instrumentation of already-running processes without restarting them. This is valuable for debugging production systems, monitoring live applications, or adding security checks dynamically. Unlike static instrumentation which works on files, runtime patching operates directly on process memory, requiring careful handling of memory protection, concurrent threads, and process state.

End-to-End Workflow

Runtime instrumentation follows an **attach → locate → generate → protect → patch → detach** sequence. The process is stopped during the critical modification phase but resumes execution with hooks active. The state machine below governs the session:



Step-by-Step Algorithm:

1. **Process Attachment and Initialization** (Runtime Patcher)
    - The user specifies a target process ID (PID) and instrumentation details.
    - The tool calls `attach_to_process(pid)` which:
        - Checks permissions (must be root or same user, subject to YAMA `ptrace_scope`)
        - Invokes `ptrace(PTRACE_ATTACH, pid, NULL, NULL)` to attach
        - Calls `wait_for_stop()` to wait for the process to enter STOPPED state
        - Saves the original register state to `runtime_session_t.original_regs` for potential restoration
    - The process is now paused; all threads are stopped.
2. **Memory Analysis and Target Location** (ELF Loader + Runtime Patcher)
    - The tool reads the process's memory mapping via `/proc/pid/maps` to locate the executable's load addresses.
    - It reads the ELF header from process memory (at the load address) to reconstruct a partial `elf_binary_t` representation.
    - Using the symbol table from the binary (or debug symbols), it resolves target function addresses in the process's address space (which may differ from file addresses due to ASLR).
    - For hook functions that need injection, it identifies suitable memory regions (using `allocate_code_cave()` on live memory via `/proc/pid/maps` analysis).
3. **Code Generation for Runtime** (Trampoline Generator)
    - Similar to static flow, the tool generates trampoline code using `generate_trampoline()`.
    - **Critical difference**: All addresses must be runtime virtual addresses, not file offsets.
    - The trampoline must be position-independent and use absolute addressing since it will be placed at a specific runtime address.
    - If injecting a hook function, the tool compiles it to PIC and resolves symbols using the process's actual loaded addresses (via `read_process_memory()` to read GOT/PLT entries).
4. **Memory Protection Modification** (Runtime Patcher)
    - Code pages are typically read-execute (`PROT_READ|PROT_EXEC`). To modify them, they must be made writable.
    - The tool calls `make_page_writable(pid, target_addr, size)` which:
        - Uses `ptrace` to inject an `mprotect` syscall into the stopped process
        - Temporarily changes the page protection to `PROT_READ|PROT_WRITE|PROT_EXEC`
        - Verifies the change succeeded by reading back the protection via `/proc/pid/smaps`
    - The same is done for any memory region where trampoline or hook code will be written.
5. **Memory Patching** (Runtime Patcher + Binary Patcher logic)
    - The tool writes the generated trampoline code into the allocated memory region using `write_process_memory()`, which calls `ptrace(PTRACE_POKEDATA)` in word-sized chunks.
    - It patches the target function prologue using `write_process_memory()` to write the jump to trampoline.
    - For each memory write, it verifies by reading back and comparing.
6. **Cache Flushing and Protection Restoration** (Runtime Patcher)
    - After code modification, the CPU's instruction cache may contain stale instructions from the old code.
    - The tool calls `flush_instruction_cache(pid, modified_addr, size)` which:
        - On x86-64, uses `ptrace` to execute a `__builtin___clear_cache` equivalent if available
        - Alternatively, relies on the fact that x86 has coherent caches for `ptrace` writes (but still calls for portability)
    - Restores original page protections (read-execute) if they were changed (optional, as some hooks keep pages writable for future modifications).
7. **Process Detachment and Resumption** (Runtime Patcher)
    - The tool calls `detach_from_process(pid)` which:
        - Optionally restores saved registers (if modified)
        - Calls `ptrace(PTRACE_DETACH, pid, NULL, NULL)`

        - The process resumes execution from where it was stopped, now with hooks active.
    - The tool cleans up local resources but leaves the injected code in the process's memory.

Data Flow Through Runtime Components:

Component	Input	Processing	Output
Runtime Patcher	PID, target function name	Attach, stop process, modify memory	Process with hooks active
ELF Loader (runtime)	Process memory regions	Parse ELF headers from memory	Partial `elf_binary_t` with runtime addresses
Trampoline Generator	Runtime addresses (target, hook)	Generate PIC trampoline	`trampoline_t*` with runtime-ready code
Memory Protector	PID, address range	Inject mprotect syscall	Memory pages with modified permissions
Memory Writer	PID, address, buffer	Write via ptrace_pokedata	Modified process memory
Cache Manager	PID, address range	Execute cache flush	CPU cache coherence

Concrete Example: Runtime Hook of `open()` in a Web Server

Consider adding logging to a running `nginx` process (PID 1234):

1. **Attach**: `attach_to_process(1234)` → stops nginx, all requests pause.
2. **Locate**: Read `/proc/1234/maps` → libc loaded at `0x7f8a5c000000`. Read ELF header, find `open` at offset `0xeb000` → runtime address `0x7f8a5c0eb000`.
3. **Generate**: `generate_trampoline(0x7f8a5c0eb000, HOOK_ADDR)` → trampoline calls `log_open`.
4. **Allocate**: Find free space in process memory (e.g., `0x7f8a5e000000` from anonymous mapping) for trampoline and hook.
5. **Protect**: `make_page_writable(1234, 0x7f8a5c0eb000, 4096)` → makes `open`'s page writable.
6. **Patch**: Write trampoline to `0x7f8a5e000000`, write jump from `open` to trampoline.
7. **Flush**: `flush_instruction_cache(1234, 0x7f8a5c0eb000, 4096)`.
8. **Detach**: `detach_from_process(1234)` → nginx resumes, now logging all file opens.

Architecture Decision: Stop-Modify-Resume vs. Continue-Modify

> **Decision: Complete Stop During Modification**
>
> - **Context**: When patching a multi-threaded process, other threads may execute code being modified, causing crashes or unpredictable behavior.
> - **Options Considered**:
>   1. **Stop all threads, modify, resume**: Use `PTRACE_ATTACH` which stops the entire process (all threads), then modify, then detach.
>   2. **Modify with threads running**: Use techniques like transactional modification or per-thread trap-and-patch, allowing some threads to continue.
> - **Decision**: Option 1 — stop the entire process during modification.
> - **Rationale**: Simplicity and safety. Stopping all threads ensures no thread executes intermediate or partially modified code. While it causes brief service interruption (typically <100ms), this is acceptable for most instrumentation scenarios. Implementing safe modification with threads running requires complex synchronization and is prone to race conditions.
> - **Consequences**: The target process experiences a brief pause (unacceptable for real-time systems). All threads stop, which may cause timeouts in connected clients. The implementation is simpler and more reliable.

Option	Pros	Cons	Chosen?
Stop all threads	Safe, simple	Process pauses, may drop connections	**Yes**
Modify with threads running	No service interruption	Extremely complex, race conditions	No

Common Pitfalls in Runtime Flow

⚠️ Pitfall: Ignoring Thread States During Attachment

- **Description**: Attaching to a multi-threaded process but only stopping the main thread, leaving worker threads running.
- **Why it's wrong**: `PTRACE_ATTACH` only stops the specific thread it attaches to. Other threads continue executing and may crash when their code is modified.
- **Fix**: After attaching to the main thread, send `SIGSTOP` to the entire process group, or attach to each thread individually. Monitor `/proc/pid/task/` for all threads.

⚠️ **Pitfall: Not Handling ASLR (Address Space Layout Randomization)**

- **Description**: Using file-based symbol addresses for runtime patching, causing patches at wrong locations.
- **Why it's wrong**: ASLR loads libraries at random addresses each run. The file offset `0x1000` might map to runtime address `0x7f8a5c001000`, not `0x4001000`.
- **Fix**: Read `/proc/pid/maps` to find load addresses, then adjust symbol offsets accordingly. Use `dlopen`/`dlsym` in an injected helper to get accurate addresses.

Interaction Patterns Between Flows

While static and runtime flows serve different use cases, they share common interaction patterns:

1. **Separation of Planning and Execution**: Both flows separate planning (`plan_patch`, `generate_trampoline`) from execution (`apply_patch`, `write_process_memory`). This allows validation before irreversible changes.
2. **ELF Representation as Central Data Structure**: Both flows rely on `elf_binary_t` as the canonical representation, though runtime flow builds it from memory rather than file.
3. **Error Handling Cascade**: Both flows implement a "fail-fast" approach: if any component fails, subsequent steps are skipped, and cleanup reverses completed operations where possible.
4. **Address Translation Layer**: Both flows need to translate between different address spaces (file ↔ virtual ↔ runtime), handled internally by components rather than exposed to users.

The tool's architecture supports both flows by having components with dual implementations: for example, memory writing is abstracted behind `write_memory()` which calls either file buffer modification or `ptrace_pokedata()` based on context.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Static Flow Coordination	Single-threaded sequential execution	Parallel patch planning with dependency analysis
Runtime Process Analysis	Reading `/proc/pid/maps` and `/proc/pid/mem`	Using `libelf` with custom memory reader
Memory Protection	`ptrace` syscall injection for `mprotect`	Using `process_vm_writev` with `PROT_GROWSDOWN`
Cache Management	Rely on x86 cache coherence	Architecture-specific intrinsics (`__builtin___clear_cache`)

Recommended File/Module Structure:

```
elflex/
├── src/
│   ├── main.c                    # CLI entry point, flow orchestration
│   ├── static_instrument.c       # Static flow implementation
│   ├── runtime_instrument.c      # Runtime flow implementation
│   ├── elfloader/                # ELF parsing (from earlier section)
│   ├── patcher/                  # Patch planning/application
│   ├── trampoline/               # Trampoline generation
│   ├── injector/                 # Code injection
│   └── runtime/                  # Ptrace and runtime utilities
├── include/
│   ├── flows.h                   # Flow coordination interfaces
│   └── ...                       # Other headers
└── test/
    ├── test_static_flow.c        # Static flow tests
    └── test_runtime_flow.c       # Runtime flow tests
```

Core Logic Skeleton for Static Flow Coordinator:

```c
// src/static_instrument.c

#include "flows.h"

#include "elfloader/elfloader.h"

#include "patcher/patcher.h"

#include "trampoline/trampoline.h"

#include "injector/injector.h"


/**

 * Execute static instrumentation flow.

 *

 * @param input_path Path to input ELF binary

 * @param output_path Path for output (patched) binary

 * @param patches Array of patch specifications from user

 * @param patch_count Number of patches

 * @param err Error structure for detailed failure reporting

 * @return 0 on success, non-zero error code on failure

 */

int execute_static_flow(const char* input_path, const char* output_path,

                        patch_spec_t* patches, size_t patch_count,

                        error_t* err) {

    elf_binary_t* binary = NULL;

    patch_site_t** planned_patches = NULL;

    trampoline_t** trampolines = NULL;

    int result = ERR_SUCCESS;



    // TODO 1: Load and parse the input binary

    // binary = elf_load(input_path);

    // if (!binary) { set_error(err, ERR_FILE_IO, "Failed to load binary"); goto cleanup; }



    // TODO 2: For each patch specification:

    //   - Resolve symbol to address if needed (elf_get_symbol_address)

    //   - Plan the patch (plan_patch) -> store in planned_patches array

    //   - If patch requires trampoline, generate it (generate_trampoline)

    //   - If hook needs injection, allocate space and prepare injection



    // TODO 3: Apply code injection if any hooks need to be added
```

```
    //   - For each hook function requiring injection:
    //     inject_section(binary, ".inject", hook_code, hook_size, INJECT_SECTION_FLAGS);


    // TODO 4: Apply all planned patches in order
    //   - For i from 0 to patch_count-1:
    //     apply_patch(binary, planned_patches[i]);


    // TODO 5: Write modified binary to output file
    //   - binary_write(binary, output_path, err);


cleanup:
    // TODO 6: Cleanup all allocated resources in reverse order
    //   - for each trampoline: trampoline_free(trampolines[i]);
    //   - for each patch: patch_free(planned_patches[i]);
    //   - if (binary) elf_binary_free(binary);
    return result;
}
```

Core Logic Skeleton for Runtime Flow Coordinator:

```c
// src/runtime_instrument.c

#include "flows.h"

#include "runtime/runtime.h"

#include "trampoline/trampoline.h"


/**

 * Execute runtime instrumentation flow.

 *

 * @param pid Target process ID

 * @param patches Array of runtime patch specifications

 * @param patch_count Number of patches

 * @param err Error structure for detailed failure reporting

 * @return 0 on success, non-zero error code on failure

 */

int execute_runtime_flow(pid_t pid, runtime_patch_spec_t* patches,

                         size_t patch_count, error_t* err) {

    runtime_session_t* session = NULL;

    trampoline_t** trampolines = NULL;

    int result = ERR_SUCCESS;



    // TODO 1: Attach to target process and stop it

    // session = create_runtime_session(pid);

    // if (attach_to_process(pid) != 0) { /* handle error */ }



    // TODO 2: Analyze process memory to locate target functions

    //   - Read /proc/pid/maps to find loaded modules

    //   - Parse ELF headers from process memory to find symbol addresses



    // TODO 3: For each patch specification:

    //   - Generate trampoline with runtime addresses

    //   - Allocate memory in target process for trampoline/hook

    //   - Change memory protections on target pages



    // TODO 4: Write trampoline code to allocated memory

    //   - write_process_memory(pid, allocated_addr, trampoline->code, trampoline->code_size);
```

```
    // TODO 5: Patch target function to jump to trampoline
    //   - make_page_writable(pid, target_addr, patch_size);
    //   - write_process_memory(pid, target_addr, jump_instructions, jump_size);

    // TODO 6: Flush instruction cache and restore protections
    //   - flush_instruction_cache(pid, target_addr, patch_size);

    // TODO 7: Detach from process and let it resume
    //   - detach_from_process(pid);

cleanup:
    // TODO 8: Cleanup local resources (but not injected code in target process)
    //   - for each trampoline: trampoline_free(trampolines[i]);
    //   - if (session) free_runtime_session(session);
    return result;
}
```

Language-Specific Hints for C:

- **Process Memory Reading**: Use `process_vm_readv()` syscall for faster bulk memory reads instead of multiple `ptrace(PTRACE_PEEKDATA)` calls.
- **Signal Handling**: When using `ptrace`, handle `SIGCHLD` to detect process state changes. Use `waitpid()` with `WNOHANG` to avoid blocking.
- **Thread Safety**: For runtime patching of multi-threaded processes, consider using `PTRACE_SEIZE` from newer Linux kernels for cleaner attachment.
- **Error Recovery**: Always save original bytes before patching, both on disk and in memory, to enable rollback on failure.

Milestone Checkpoint for Flow Integration:

After implementing both flows, test with a simple program:

```bash
# Build test target
cat > test_prog.c << 'EOF'
#include <stdio.h>
void hello() { printf("Hello\n"); }
int main() { hello(); return 0; }
EOF
gcc -o test_prog test_prog.c

# Test static flow
./elflex static --binary test_prog --hook hello --hook-func my_hook --output test_prog_hooked
./test_prog_hooked  # Should call hook

# Test runtime flow
./test_prog &  # Run in background
PID=$!
./elflex runtime --pid $PID --hook hello --hook-func my_hook
# Process should now output hook messages
```

Debugging Tips for Flow Issues:

Symptom	Likely Cause	How to Diagnose	Fix
Static-patched binary crashes on start	Corrupted ELF headers	Run `readelf -h patched_binary` and compare with original	Ensure `binary_write()` updates all header fields correctly
Runtime patch causes immediate segfault	Wrong jump target address	Use `gdb attach` to disassemble at patched address	Verify address calculation accounts for ASLR
Hook called multiple times recursively	Trampoline doesn't preserve return address	Examine trampoline assembly for proper stack handling	Ensure register save/restore includes RSP management
Process hangs after `ptrace` attach	Not all threads stopped	Check `/proc/pid/task/*/status` for thread states	Attach to or signal-stop all threads
Modified code not executed (old code runs)	Instruction cache not flushed	Check if `__builtin___clear_cache` is called	Explicitly call cache flush or use `msync`

Error Handling and Edge Cases

> **Milestone(s):** 1 (Instruction-level Patching), 2 (Function Hooking with Trampolines), 3 (Code Injection), 4 (Runtime Patching via ptrace)

Building a binary instrumentation tool is fundamentally an exercise in managing complexity and failure. Unlike applications that operate on well-defined data formats, ELFlex manipulates the very foundation of program execution—machine code and binary formats—where even minor errors can cause catastrophic failures like segmentation faults, infinite loops, or corrupted output files. This section catalogs the predictable failure modes across all components and establishes robust strategies for detection, recovery, and cleanup. The philosophy is **"fail early, fail cleanly"**: detect errors as soon as possible, provide clear diagnostic information, and ensure the system never leaves the target binary or process in an unrecoverable state.

Common Failure Modes and Detection

Binary instrumentation failures can be categorized by their origin: malformed input, invalid operations, system resource limitations, and runtime environment constraints. The following table enumerates the most critical failure modes, their detection mechanisms, and the component where they primarily occur.

Failure Mode	Primary Component(s)	Detection Method	Error Code (Example)	Description
Malformed ELF file	ELF Loader	Signature validation, header field sanity checks, section/segment consistency validation	`ERR_INVALID_ELF`	Input file does not conform to ELF64 specification: missing magic bytes, inconsistent header sizes, section offsets beyond file size, or overlapping segments.
File I/O errors	ELF Loader, Binary Writer	Return value checking from `read()`, `write()`, `open()`, `fstat()` system calls	`ERR_FILE_IO`	Cannot read input file (permission denied, file not found), cannot write output file (disk full, path read-only), or partial read/write operations.
Out of memory	All components	Checking return value of `malloc()`, `calloc()`, `realloc()`; monitoring memory usage during large operations	`ERR_OUT_OF_MEMORY`	System or process memory exhausted while loading binary, allocating patch structures, or generating trampoline code.
Invalid offset for patching	Patch Planner	Validating target offset resides within an executable section (`.text` or other with `SHF_EXECINSTR` flag), ensuring offset is at an instruction boundary	`ERR_INVALID_OFFSET`	User requests patching at an address not in executable code, or at a misaligned address that would split a multi-byte instruction.
Patch overlap	Patch Planner	Maintaining a registry of already-modified regions within the binary and checking new patches don't intersect	`ERR_PATCH_OVERLAP`	Attempting to patch an instruction that has already been modified (e.g., trying to hook a function twice).
Symbol not found	ELF Loader (symbol resolution), Trampoline Generator	Searching symbol table (`.symtab` / `.dynsym`) and string table for exact match; handling weak symbols	`ERR_SYMBOL_NOT_FOUND`	Requested function name does not exist in binary's symbol table, or symbol exists but has no defined address (e.g., undefined external reference).
Section not found	ELF Loader, Code Injector	Iterating through section headers and comparing names from string table	`ERR_SECTION_NOT_FOUND`	Required standard section (`.text`, `.symtab`) is missing, or user references a custom section that doesn't exist.
Insufficient space for injection	Code Injector	Checking if code cave size meets requirement, or if adding new segment would cause virtual address overlap	Custom error (e.g., `ERR_INJECT_NO_SPACE`)	No suitable unused region (`code cave`) large enough for injected code, and binary cannot be expanded due to address space constraints.
RIP-relative fixup failure	Trampoline Generator	Disassembling relocated prologue instruction; detecting RIP-relative addressing mode; validating new offset fits in 32-bit signed field	Custom error (e.g., `ERR_RIP_FIXUP`)	Relocated instruction with RIP-relative addressing (like `mov rax, [rip+0x1234]`) cannot be adjusted because new offset exceeds ±2GB range.
Ptrace attach permission denied	Runtime Patcher	Checking `errno` after `ptrace(PTRACE_ATTACH, ...)` call (EACCES, EPERM)	Custom error (e.g., `ERR_PTRACE_PERM`)	Target process is not owned by same user (without CAP_SYS_PTRACE), or YAMA `ptrace_scope` restricts operation.

Failure Mode	Primary Component(s)	Detection Method	Error Code (Example)	Description
Process not found	Runtime Patcher	Checking `kill(pid, 0)` or `ptrace` attach fails with ESRCH	Custom error (e.g., `ERR_PROCESS_NOT_FOUND`)	Specified PID does not correspond to a running process.
Memory protection change failure	Runtime Patcher	Checking return value of injected `mprotect` syscall (via `PTRACE_GETREGS` / `PTRACE_SETREGS`)	Custom error (e.g., `ERR_MPROTECT_FAILED`)	Cannot make code page writable due to kernel security policy (e.g., PaX/grsecurity), or invalid address.
Multi-threaded process complications	Runtime Patcher	Detaching from process but other threads may be in inconsistent state; attempting to freeze all threads via `PTRACE_SEIZE`	Custom error (e.g., `ERR_THREADS_UNSAFE`)	Process has multiple threads; patching one thread's code while others execute leads to race conditions and crashes.
ASLR complications	Runtime Patcher	Reading `/proc/pid/maps` to get actual load addresses; comparing to binary's virtual addresses	Custom error (e.g., `ERR_ASLR_MISMATCH`)	Runtime addresses differ from static binary addresses due to Address Space Layout Randomization; offsets need adjustment.
Instruction cache coherence	Runtime Patcher	Architecture-specific; failure manifests as old code executing despite patch	Detection via verification read after write	After writing trampoline code, CPU may still execute stale instructions from cache; explicit flush required (e.g., `__builtin___clear_cache`).

> **Key Insight:** Many errors in binary instrumentation are **latent**—they don't cause immediate failure in the tool itself, but rather manifest later when the modified binary is executed. This makes comprehensive validation at instrumentation time critical. For example, an invalid patch offset might not crash the patching tool, but will certainly crash the patched binary when execution reaches the corrupted instruction.

Detection Strategy Implementation: Each component should validate its inputs and operations aggressively. The `error_t` structure provides a uniform way to capture error context. Functions should return a status code (0 for success, non-zero error code) and populate an `error_t*` parameter when provided. This allows error information to propagate up the call stack while maintaining function signatures that match the naming conventions.

Recovery and Cleanup Strategy

When errors occur, ELFlex must clean up allocated resources and, when possible, leave the system in a consistent state. The cleanup strategy follows a **"ownership hierarchy"** where parent structures are responsible for freeing their owned child resources. For runtime patching, the strategy emphasizes **"minimal disruption"**—if we cannot complete the patching, we must restore the process to its original state before detaching.

1. Static Instrumentation Cleanup (Milestones 1-3)

The static instrumentation workflow (on-disk binary modification) follows a clear resource ownership chain:

```
elf_binary_t (owns) → sections/segments data
                    → patch_site_t* (if any)
                    → trampoline_t* (if any)
                    → injected_section_t* (if any)
```

When an error occurs during static instrumentation:

Phase	Cleanup Actions	Rationale
During ELF loading	If `read_file()` or header parsing fails, free any partially allocated buffers and close file descriptors. The `elf_binary_t*` itself is not returned, so caller has nothing to clean.	Prevents memory leaks from partial loads.
During patch planning	If `plan_patch()` fails (invalid offset, overlap), it returns `NULL`. Any intermediate allocations within the function must be freed before returning.	Patch not created, so no cleanup required at caller level.
During trampoline generation	If `generate_trampoline()` fails (e.g., RIP fixup impossible), it must free any partially assembled code buffers before returning `NULL`.	Prevents memory leaks of partial trampoline code.
During code injection	If `inject_section()` fails (no space, relocation error), it must free the `injected_section_t` and any allocated dependency arrays, and revert any header modifications to the `elf_binary_t`.	Binary must remain in valid state; injection is atomic—either fully succeeds or fully fails.
During binary writing	If `binary_write()` fails (disk full, I/O error), the in-memory `elf_binary_t` remains unchanged. Caller can decide to retry or abort. Output file should be deleted if partially written.	Prevents corrupted output files; in-memory model remains intact for debugging.
Overall failure	The top-level instrumentation function should call `elf_binary_free()`, which recursively frees all owned resources: patches, trampolines, and injected sections via their respective `*_free()` functions.	Single cleanup point ensures no resource leaks regardless of where failure occurred.

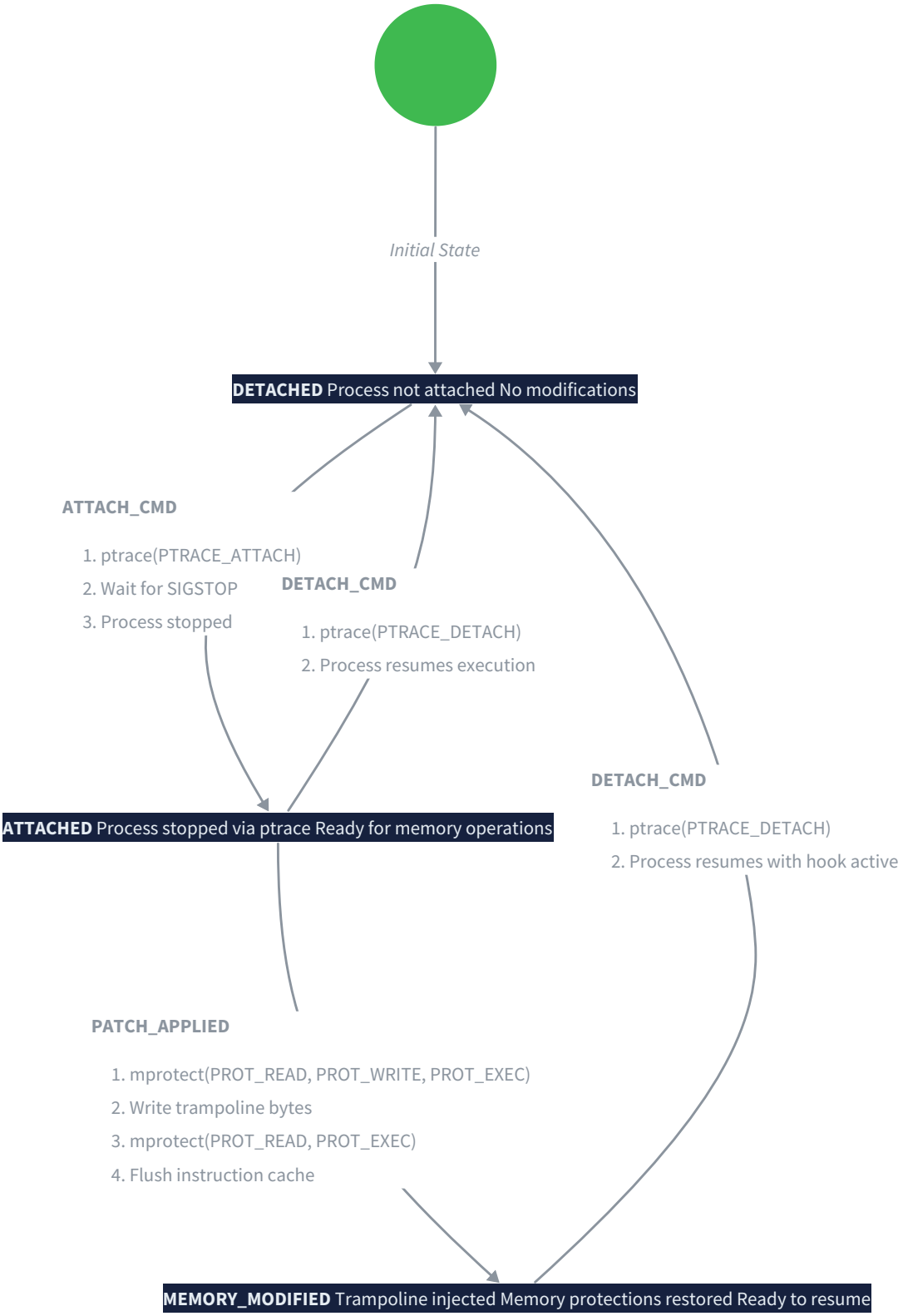
ADR: Error Handling Approach — Return Codes with Optional Error Context

> **Decision: Use Return Codes with Optional Error Context Parameter**
>
> - **Context**: Need to signal success/failure from functions while providing detailed error information for debugging. Must work with C's limited error handling mechanisms.
> - **Options Considered**:
>   1. **Return codes only** (int with errno): Simple, but loses context-specific error messages.
>   2. **Global error state** (like `errno` with custom functions): Thread-unsafe, mixes errors from different operations.
>   3. **Error struct passed by reference**: Provides rich context, caller-controlled, but adds parameter overhead.
> - **Decision**: Use return codes (0 success, non-zero error codes defined as constants) with an optional `error_t*` parameter that functions can populate if non-NULL.
> - **Rationale**: Balances simplicity with diagnostic power. Callers that don't need details can pass NULL; those that do get structured error messages with context. Thread-safe because error state is passed explicitly.
> - **Consequences**: All functions in the public API need an `error_t*` parameter (often last). Callers must check return codes and optionally inspect error_t. Adds some verbosity but improves debuggability.

Option	Pros	Cons	Chosen?
Return codes only	Simple API, no extra parameters	No diagnostic details, forces guesswork	No
Global error state	Easy to use, no parameter passing	Not thread-safe, error state overwritten	No
Error struct by reference	Rich diagnostics, thread-safe, caller controls lifetime	More verbose API, caller must manage error_t	**Yes**

2. Runtime Patching Cleanup (Milestone 4)

Runtime patching introduces critical cleanup requirements because we're modifying a live process. The cleanup strategy follows a **state machine** (refer to



) that ensures we never leave the target process in a corrupted or stuck state.

> **Mental Model: The Surgical Rollback Protocol** Imagine a surgical team that begins a delicate operation. Before making any irreversible cuts, they prepare a "rollback plan": mark the incision site, keep original tissue samples, and have reversal procedures ready. If complications arise, they immediately execute the rollback, stitching back exactly as before. Similarly, the runtime patcher must save the original bytes of any code it plans to modify before writing trampolines. If any step fails, it restores the original code and protections before detaching.

Runtime Patching State Transitions and Cleanup:

Current State	Event (Error)	Cleanup Actions	Next State
ATTACHED (process stopped)	`attach_to_process()` fails (permission, no process)	No process state changed; free local session resources.	DETACHED
ATTACHED	`make_page_writable()` fails (mprotect error)	Restore original page protections (if changed), detach from process.	DETACHED
ATTACHED	`write_process_memory()` fails (partial write)	Restore original code bytes from backup, restore page protections, detach.	DETACHED
MEMORY_MODIFIED (code written)	Any subsequent error (cache flush fail, detach error)	We cannot safely rollback after cache flush? Actually we can still restore code, but execution may have already hit patched code. Best effort: restore original code, log warning, detach.	DETACHED
Any state	Signal interference (process receives signal while stopped)	Save signal info, continue process with signal delivered after detach.	DETACHED

Key Recovery Procedures:

1. **Code Modification Rollback:**
   - Before overwriting any code, read and save the original bytes (enough for the trampoline jump).
   - Store these bytes in the `trampoline_t` or session context.
   - If any step fails after modification, write back the saved bytes using `write_process_memory()`.
   - This ensures the process resumes with its original code intact.
2. **Page Protection Restoration:**
   - After successfully making a page writable (`mprotect(PROT_READ|PROT_WRITE|PROT_EXEC)`), save the original protection flags.
   - During cleanup (success or failure), restore original flags via another `mprotect` call.
   - Critical for security and stability—leaving code pages writable makes the process vulnerable to exploitation.
3. **Process State Preservation:**
   - When attaching via `ptrace(PTRACE_ATTACH)`, the process is stopped. We must save its register state (`PTRACE_GETREGS`) before any modification.
   - If we inject a syscall (for `mprotect`), we must ensure registers are restored to original values before detaching.
   - Any pending signals should be collected via `PTRACE_GETSIGINFO` and re-injected upon detach (`PTRACE_CONT` with signal).
4. **Multi-threaded Process Consideration:**
   - In advanced scenarios, patching a multi-threaded process requires stopping all threads (`PTRACE_SEIZE` with `PTRACE_INTERRUPT`).
   - If we cannot stop all threads, we should abort the operation early (before any modification) to avoid race conditions.
   - Cleanup must resume all threads if they were stopped.

3. Memory Management and Ownership

To prevent resource leaks, every allocated resource must have a clear owner responsible for its deallocation. The following ownership hierarchy is enforced:

- `elf_binary_t*` returned by `elf_load()` owns:
  - The file buffer (`buffer`)

    - Array of `section_t*` (each owning its `name` and `data`)
    - Array of `segment_t*` (each owning its `data`)
    - Any `patch_site_t*` added via `plan_patch()` (owned via array)
    - Any `injected_section_t*` added via `inject_section()` (owned via array)
- `patch_site_t*` owns:
    - `new_bytes` buffer
    - `nop_padding` buffer
- `trampoline_t*` owns:
    - `code` buffer
    - `original_prologue` buffer
    - `relocated_prologue` buffer
- `injected_section_t*` owns:
    - `name` string
    - `data` buffer
    - Array of `symbol_dependencies` strings
- `runtime_session_t*` owns:
    - Saved register state arrays
    - Saved original code bytes

The `*_free()` functions (`elf_binary_free`, `patch_free`, `trampoline_free`, `injected_section_free`) recursively free owned resources. This design ensures that a single call to `elf_binary_free()` cleans up the entire instrumentation state, even if intermediate operations failed partially.

4. Error Reporting and User Experience

When errors occur, ELFlex should provide actionable information:

- **Console output**: Clear error messages stating what failed and why, using the `error_t.message` and `error_t.context` fields.
- **Verbose logging**: Optional debug mode that logs each step, useful for diagnosing complex failures like RIP-relative fixup errors.
- **Error codes**: Consistent use of predefined `ERR_*` constants allows scripted error handling.
- **Partial success handling**: For batch operations (patch multiple functions), consider continuing after non-fatal errors and reporting a summary.

> **Critical Principle:** Never proceed with uncertainty. If the tool cannot guarantee the binary or process will be left in a valid state, it must abort and clean up. A failed instrumentation is better than a silently corrupted binary.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Error structure	Fixed-size `char` arrays in `error_t`	Dynamically allocated strings for unlimited message length
Error propagation	Manual checking and forwarding of `error_t*`	Macro-based error propagation (like `ERR_PROPAGATE`)
Memory debugging	Valgrind, AddressSanitizer	Custom allocator with tracking, leak detection at shutdown
Runtime error detection	Check `errno` after system calls	Use `strace`-like interception to log all syscalls during patching

B. Recommended File/Module Structure:

```
elflex/
  include/elflex/
    error.h          ← Error codes and error_t structure
  src/
    error.c          ← Error creation, formatting, freeing
    elf_loader.c     ← ELF loading with error handling
    patch.c          ← Patching with error handling
    trampoline.c     ← Trampoline generation with error handling
    inject.c         ← Code injection with error handling
    runtime.c        ← Runtime patching with error handling
    utils/           ← Helper functions with error handling
      io.c           ← read_file, write_file with error reporting
      memory.c       ← Safe allocation wrappers
  tests/
    test_error.c     ← Unit tests for error handling
    test_invalid_elf.c ← Test handling of malformed binaries
```

C. Infrastructure Starter Code (Complete Error Handling Module):

```c
// include/elflex/error.h

#ifndef ELFLEX_ERROR_H

#define ELFLEX_ERROR_H


#include <stddef.h>


#define ERROR_MSG_SIZE 256

#define ERROR_CTX_SIZE 128


typedef struct {

    int code;

    char message[ERROR_MSG_SIZE];

    char context[ERROR_CTX_SIZE];

} error_t;


// Error codes (subset shown)

#define ERR_SUCCESS 0

#define ERR_INVALID_ELF 1

#define ERR_FILE_IO 2

#define ERR_OUT_OF_MEMORY 3

#define ERR_SECTION_NOT_FOUND 4

#define ERR_SYMBOL_NOT_FOUND 5

#define ERR_PATCH_OVERLAP 6

#define ERR_INVALID_OFFSET 7

#define ERR_PTRACE_PERM 100

#define ERR_MPROTECT_FAILED 101


// Public API

error_t* error_create();

void error_free(error_t* err);

void error_set(error_t* err, int code, const char* message, const char* context);

void error_clear(error_t* err);

const char* error_code_to_string(int code);


#endif // ELFLEX_ERROR_H
```

```c
// src/error.c

#include <elflex/error.h>

#include <string.h>

#include <stdlib.h>


error_t* error_create() {

    error_t* err = calloc(1, sizeof(error_t));

    if (!err) return NULL;

    return err;

}


void error_free(error_t* err) {

    if (err) free(err);

}


void error_set(error_t* err, int code, const char* message, const char* context) {

    if (!err) return;



    err->code = code;



    // Safely copy message

    if (message) {

        strncpy(err->message, message, ERROR_MSG_SIZE - 1);

        err->message[ERROR_MSG_SIZE - 1] = '\0';

    } else {

        err->message[0] = '\0';

    }



    // Safely copy context

    if (context) {

        strncpy(err->context, context, ERROR_CTX_SIZE - 1);

        err->context[ERROR_CTX_SIZE - 1] = '\0';

    } else {

        err->context[0] = '\0';

    }

}
```

```
void error_clear(error_t* err) {
    if (err) {
        err->code = ERR_SUCCESS;
        err->message[0] = '\0';
        err->context[0] = '\0';
    }
}

const char* error_code_to_string(int code) {
    switch (code) {
        case ERR_SUCCESS: return "Success";
        case ERR_INVALID_ELF: return "Invalid ELF file";
        case ERR_FILE_IO: return "File I/O error";
        case ERR_OUT_OF_MEMORY: return "Out of memory";
        case ERR_SECTION_NOT_FOUND: return "Section not found";
        case ERR_SYMBOL_NOT_FOUND: return "Symbol not found";
        case ERR_PATCH_OVERLAP: return "Patch overlaps existing modification";
        case ERR_INVALID_OFFSET: return "Invalid offset (not in executable code)";
        case ERR_PTRACE_PERM: return "Ptrace permission denied";
        case ERR_MPROTECT_FAILED: return "Failed to change memory protection";
        default: return "Unknown error";
    }
}
```

D. Core Logic Skeleton Code (Error-Aware Function):

```c
// Example: Modified elf_load with error handling

#include <elflex/error.h>

#include <elflex/elf_loader.h>


elf_binary_t* elf_load(const char* filename, error_t* err) {

    if (!filename) {

        if (err) error_set(err, ERR_FILE_IO, "Filename is NULL", "elf_load");

        return NULL;

    }



    // TODO 1: Get file size using stat(), handle errors (check errno)

    //   If stat fails: error_set(err, ERR_FILE_IO, "Cannot stat file", filename); return NULL



    // TODO 2: Allocate buffer for entire file

    //   If malloc fails: error_set(err, ERR_OUT_OF_MEMORY, "Failed to allocate file buffer", filename); return NULL



    // TODO 3: Read file into buffer using open()/read()

    //   If open/read fails: error_set(err, ERR_FILE_IO, "Failed to read file", filename); free buffer; return NULL



    // TODO 4: Validate ELF magic bytes using is_valid_elf()

    //   If invalid: error_set(err, ERR_INVALID_ELF, "Invalid ELF magic", filename); free buffer; return NULL



    // TODO 5: Parse ELF headers, check consistency

    //   If e_phnum * e_phentsize > file size: error_set(err, ERR_INVALID_ELF, "Program header table out of bounds",
filename); free buffer; return NULL



    // TODO 6: Allocate and populate elf_binary_t structure

    //   If any allocation fails: error_set(err, ERR_OUT_OF_MEMORY, "Failed to allocate ELF structure", filename); free
all intermediate allocations; return NULL



    // TODO 7: Parse section headers, load sections

    //   If section offset+size > file size: error_set(err, ERR_INVALID_ELF, "Section data out of bounds", section name);
elf_binary_free(binary); return NULL



    // TODO 8: Return successfully loaded binary

    //   if (err) error_clear(err);  // Clear any previous error on success

    //   return binary;
```

```c
    return NULL; // placeholder
}

// Example: Patch application with rollback on failure
int apply_patch(elf_binary_t* binary, patch_site_t* patch, error_t* err) {
    if (!binary || !patch) {
        if (err) error_set(err, ERR_INVALID_OFFSET, "Invalid arguments", "apply_patch");
        return -1;
    }

    // TODO 1: Validate patch target is within .text section bounds
    //   If out of bounds: error_set(err, ERR_INVALID_OFFSET, "Patch offset outside .text", NULL); return -1

    // TODO 2: Check if this region is already patched (maintain patch registry in binary)
    //   If overlapping: error_set(err, ERR_PATCH_OVERLAP, "Patch overlaps existing modification", NULL); return -1

    // TODO 3: Save original bytes for potential rollback
    //   Copy original_size bytes from binary->buffer + offset to patch->original_bytes (if not already saved)

    // TODO 4: Apply new_bytes to the binary buffer
    //   memcpy(binary->buffer + offset, patch->new_bytes, patch->new_size);

    // TODO 5: Apply NOP padding if needed
    //   if (patch->nop_padding_size > 0) memcpy(binary->buffer + offset + patch->new_size, ...);

    // TODO 6: Update patch registry to mark this region as modified
    //   Add patch to binary->patches array

    // TODO 7: Mark patch as applied
    //   patch->applied = 1;

    // TODO 8: On success: if (err) error_clear(err); return 0

    return -1; // placeholder
```

```
}
```

E. Language-Specific Hints (C):

- **System call error checking**: Always check return values of system calls and library functions. Use `perror()` for quick debugging, but for production, populate `error_t` with `strerror(errno)`.
- **Memory allocation wrappers**: Create helper functions like `safe_malloc(size, err)` that check for NULL and set error appropriately.
- **Cleanup with `goto`**: In complex C functions with multiple allocation points, use `goto cleanup` pattern for centralized error handling:

```c
int complex_operation(error_t* err) {
    char* buf1 = NULL;
    char* buf2 = NULL;

    buf1 = malloc(100);
    if (!buf1) { error_set(err, ERR_OUT_OF_MEMORY, ...); goto cleanup; }

    buf2 = malloc(200);
    if (!buf2) { error_set(err, ERR_OUT_OF_MEMORY, ...); goto cleanup; }

    // ... operation ...

    cleanup:
    free(buf1);
    free(buf2);
    return (err && err->code != ERR_SUCCESS) ? -1 : 0;
}
```

- **Ptrace error handling**: After `ptrace()` calls, check `errno` and use `PTRACE_GETERRNO` on some systems to get target's errno.

F. Milestone Checkpoint for Error Handling:

Milestone 1 Test: Create a test that attempts to patch at invalid offset:

```
$ ./elflex patch --offset 0xDEADBEEF test.bin
Error: Invalid offset (not in executable code) [context: apply_patch]
```

Verify the tool exits cleanly with non-zero code, no memory leaks (check with Valgrind), and the original binary is unchanged.

Milestone 4 Test: Attempt to attach to a non-existent process:

```
$ ./elflex runtime --pid 99999 --hook malloc
Error: Process not found [context: attach_to_process]
```

Verify the tool doesn't crash and exits cleanly.

G. Debugging Tips for Error Scenarios:

Symptom	Likely Cause	How to Diagnose	Fix
Segmentation fault after loading patched binary	Corrupted ELF headers during modification	Run `readelf -h patched.bin` and compare with original. Check section offsets.	Ensure `binary_write()` updates all header fields correctly, especially `e_shoff` if sections moved.
Hook not called, function executes normally	Trampoline jump not installed properly	Disassemble patched binary at function entry: `objdump -d patched.bin | grep -A 10 '<target_func>:'`. Should see jump.	Verify trampoline generation produces correct relative/absolute jump. Check jump offset calculation.
Process hangs after ptrace attach	Process stopped but not resumed properly	Use `strace -p <pid>` to see if process is in stopped state. Check for pending signals.	Ensure `detach_from_process()` calls `PTRACE_CONT` or `PTRACE_DETACH`. Handle saved signals properly.
"Permission denied" on ptrace attach	YAMA ptrace_scope restriction or lack of CAP_SYS_PTRACE	Check `cat /proc/sys/kernel/yama/ptrace_scope`. If 1, only parents can ptrace.	Run as root, or use `sudo`, or adjust YAMA settings (security risk).
Binary grows enormously after injection	Segment alignment causing large padding	Check `readelf -l patched.bin` to see segment alignments (0x200000 common).	Use `allocate_code_cave()` to find existing space rather than always adding new aligned segment.
Modified binary fails with "ELF file type unknown"	`e_type` field corrupted	Compare `readelf -h original.bin` and `patched.bin` `e_type` field.	Ensure patching doesn't overwrite ELF header. Keep header region read-only in modifications.

Testing Strategy

> **Milestone(s):** 1 (Instruction-level Patching), 2 (Function Hooking with Trampolines), 3 (Code Injection), 4 (Runtime Patching via ptrace)

Building a binary instrumentation tool is inherently complex and error-prone—like performing surgery on a living organism. Each component must be rigorously tested both in isolation and as part of an integrated system. This section outlines a **defense-in-depth testing strategy** that progresses from unit tests of individual parsers to full-system integration tests with real binaries. The approach mirrors the tool's own architecture: starting with foundational components and building upward, with each milestone serving as a natural testing checkpoint.

Testing Approach and Tools

Mental Model: The Surgical Simulator and Flight Recorder

Think of testing ELFlex as analogous to training a surgical team. Before operating on a real patient, surgeons practice on **simulated tissue** (unit tests for individual components) and **cadavers** (integration tests with simple binaries). During actual operations, they use **vital sign monitors** (debugging tools like `gdb` and `strace`) to track the patient's state. Finally, a **flight recorder** (comprehensive logging within ELFlex itself) captures every step of the procedure for post-mortem analysis if something goes wrong.

This multi-layered approach ensures that errors are caught at the earliest possible stage, from malformed ELF parsing to subtle trampoline register corruption. The testing pyramid for ELFlex consists of:

1. **Unit Tests** - Validate individual functions and data structures in isolation
2. **Component Integration Tests** - Test interactions between 2-3 components
3. **End-to-End System Tests** - Full pipeline from binary input to patched output
4. **Runtime Behavior Verification** - Execute patched binaries and confirm correct behavior

The following table outlines the primary testing tools and their roles in this strategy:

Tool	Purpose in Testing	Example Usage
`readelf`	Inspect ELF structure before/after modification	Verify section headers, segment alignment, symbol tables
`objdump -d`	Disassemble code to verify instruction patches	Confirm trampoline placement, NOP sled insertion
`gdb`	Runtime verification and debugging	Single-step through trampolines, inspect registers, set breakpoints
`strace`	Monitor system calls during runtime patching	Verify `ptrace` operations, `mprotect` calls
`diff`	Compare binary outputs	Ensure only intended changes were made
Custom Test Harness	Automated test execution and verification	Run test suite, capture output, compare against expected results
`valgrind`	Memory leak detection	Validate cleanup of `elf_binary_t`, `trampoline_t` structures
AddressSanitizer (ASan)	Detect buffer overflows, use-after-free	Instrument test builds to catch memory corruption early

Testing Infrastructure Philosophy

The testing approach follows three core principles:

> **Principle 1: Test with Known-Simple Binaries First** Begin testing with trivial, purpose-built test binaries that have minimal dependencies and predictable behavior. These "laboratory specimens" eliminate variables and make failures easier to diagnose.

> **Principle 2: Preserve Original Test Assets** Never modify original test binaries in place. Always work on copies, and maintain checksums of originals to verify that only intended changes occur.

> **Principle 3: Test Both Success and Failure Paths** For each component, test not only the happy path but also error conditions (malformed input, invalid offsets, permission errors) to ensure robust error handling.

The following architecture decision record formalizes the choice between simple test binaries and complex real-world applications as primary test targets:

Decision: Use Purpose-Built Test Binaries Over Complex Real Applications

- **Context**: We need reliable, reproducible tests that can run quickly and fail with clear diagnostics. Real applications (like `ls` or `bash`) are large, complex, and their behavior may vary between systems.
- **Options Considered**:
  1. **Real system binaries**: Test on `/bin/ls`, `/usr/bin/date`, etc.
  2. **Purpose-built test binaries**: Create small C programs with specific, observable behaviors.
  3. **Hybrid approach**: Use both, starting with simple binaries and progressing to complex ones.
- **Decision**: Option 2 (Purpose-built test binaries) as primary test suite, with Option 3 for advanced integration testing.
- **Rationale**:
  - Purpose-built binaries are **portable** (no system dependencies)
  - They have **deterministic, observable behavior** (e.g., printing specific strings)
  - They can be **minimized** to isolate specific test cases (e.g., a binary with exactly one function to hook)
  - They eliminate variables like ASLR, library versions, and system configuration differences
- **Consequences**:
  - ✅ Easy to create and maintain test suite
  - ✅ Fast test execution
  - ✅ Clear failure diagnostics
  - ❌ May not catch edge cases present in real, complex binaries
  - ❌ Requires building and maintaining test binaries alongside tool

Option	Pros	Cons	Selected
Real system binaries	Tests real-world scenarios, catches complex edge cases	Non-portable, unpredictable, may require root permissions, slow	No (secondary only)
Purpose-built binaries	Portable, deterministic, fast, clear diagnostics	May miss real-world complexity	**Yes** (primary)
Hybrid approach	Combines benefits of both	More complex test infrastructure, slower overall	Yes (supplementary)

Milestone Checkpoints

Each milestone represents a natural testing boundary with specific acceptance criteria. The following checkpoints provide concrete, actionable testing procedures that verify each milestone's functionality end-to-end.

Milestone 1 Checkpoint: Instruction-level Patching

What to Test: The ability to load an ELF binary, locate its `.text` section, replace instructions at specified offsets, and produce a valid patched binary that executes with modified behavior.

Pre-Test Setup: Create a simple test binary (`test_m1.c`) with observable behavior:

```c
// test_m1.c - Simple test for instruction patching
#include <stdio.h>
#include <unistd.h>

void target_function() {
    printf("Original behavior\n");
    fflush(stdout);
}

int main() {
    target_function();
    return 0;
}
```

Compile with debugging symbols and no optimizations:

```bash
gcc -g -O0 -no-pie -o test_m1 test_m1.c
```

Test Procedure:

1. **Locate Patch Target**: Use `objdump -d test_m1` to find the address/offset of an instruction in `target_function` . For example, find the `printf` call instruction.
2. **Create Patch File**: Write a simple patch specification (JSON or custom format) indicating:
   - Target offset (file offset of instruction)
   - Replacement bytes (e.g., `NOP` sled or different instruction)
3. **Run ELFlex in Patch Mode**: Execute the tool with the test binary and patch specification:

```bash
./elflex patch --binary test_m1 --patch patch_spec.json --output test_m1_patched
```

4. **Verify Binary Integrity**: Use `readelf` to ensure the ELF structure remains valid:

```bash
readelf -h test_m1_patched  # Should show valid ELF header

readelf -S test_m1_patched  # Sections should be intact
```

5. **Verify Instruction Change**: Disassemble the patched area:

```bash
objdump -d test_m1_patched --start-address=<function_addr> --stop-address=<function_addr+20>
```

Confirm the target instruction has been replaced.

6. **Test Execution**: Run the patched binary and observe behavior:

```bash
./test_m1_patched
```

If patched with `NOP` sled, the program should run but potentially skip the `printf` (resulting in no output or different output).

Expected Output Matrix:

Test Case	Expected Result	Verification Command
Valid offset in .text	Patch succeeds, binary executes	`./test_m1_patched` runs without crash
Offset outside .text	Returns `ERR_INVALID_OFFSET`	Tool returns error code 7
Replacement larger than original (with NOP padding)	Patch succeeds, padded with NOPs	`objdump` shows NOPs after patch
Malformed ELF input	Returns `ERR_INVALID_ELF`	Tool returns error code 1
Binary with stripped symbols	Patch still works (uses offsets, not symbols)	Tool accepts raw offset parameter

Success Criteria:

- ✅ Patched binary loads and executes without segmentation fault
- ✅ `readelf` reports valid ELF structure
- ✅ `objdump` shows the intended instruction replacement
- ✅ Original test binary remains unmodified (checksum unchanged)
- ✅ Error cases are handled gracefully with clear error messages

Milestone 2 Checkpoint: Function Hooking with Trampolines

What to Test: The ability to detect function entry points via symbol table, replace function prologue with jump to trampoline, execute hook function, and continue to original function.

Pre-Test Setup: Create a test binary with multiple functions and observable side effects:

```c
// test_m2.c - Test function hooking with arguments

#include <stdio.h>

#include <unistd.h>


int multiply(int a, int b) {

    printf("Multiplying %d * %d = ", a, b);

    int result = a * b;

    printf("%d\n", result);

    return result;

}


int main() {

    int x = multiply(5, 7);

    printf("Result: %d\n", x);

    return 0;

}
```

Compile with symbols preserved:

```bash
gcc -g -O0 -no-pie -o test_m2 test_m2.c
```

Also create a hook function to inject:

```c
// hook_m2.c - Hook function to intercept multiply

int hook_multiply(int a, int b) {

    printf("HOOK: Arguments: a=%d, b=%d\n", a, b);

    // Let original function execute with modified arguments

    return 0; // We'll call original from trampoline

}
```

Compile as position-independent code:

```bash
gcc -g -O0 -fPIC -c hook_m2.c -o hook_m2.o
```

Test Procedure:

1. **Extract Symbol Address**: Use `readelf -s test_m2 | grep multiply` to find the address of `multiply` function.
2. **Generate Trampoline**: Run ELFlex in trampoline generation mode:

```bash
./elflex trampoline --binary test_m2 --function multiply --hook hook_m2.o --output test_m2_hooked
```

3. **Verify Trampoline Placement**:
    - Use `readelf -S test_m2_hooked` to check for new sections (e.g., `.inject`)
    - Use `objdump -d test_m2_hooked` to examine:
        - The prologue of `multiply` should start with a `jmp` to trampoline

- The trampoline code should include register saves, hook call, and jump to original function body

4. **Test Execution with Hook**:

```bash
./test_m2_hooked
```

Expected output:

```
HOOK: Arguments: a=5, b=7
Multiplying 5 * 7 = 35
Result: 35
```

5. **Test Argument Modification**: Modify the hook to change arguments (e.g., `a = a * 2`) and verify the multiplication uses modified values.
6. **Test RIP-Relative Fixup**: Create a test function using RIP-relative addressing (e.g., loading a global variable) to ensure relocated prologue instructions work correctly.

Expected Output Matrix:

Test Case	Expected Result	Verification Method
Hook on function with symbol	Trampoline inserted, hook called	Hook message appears before original output
Hook on function without public symbol (static)	Returns `ERR_SYMBOL_NOT_FOUND`	Tool returns error code 5
Hook with argument modification	Original function receives modified arguments	Output shows modified calculation
Function with RIP-relative addressing in prologue	Relocated prologue executes correctly	No segmentation fault, correct value loaded
Multiple hooks on same binary	All hooks fire in correct order	All hook messages appear

Success Criteria:

- ✅ Hook function executes before original function
- ✅ Original function behavior preserved (with potentially modified arguments)
- ✅ No register corruption (original function returns correct value)
- ✅ Stack integrity maintained (no crashes)
- ✅ RIP-relative instructions in relocated prologue work correctly
- ✅ Multiple hooks can be installed independently

Milestone 3 Checkpoint: Code Injection

What to Test: The ability to inject new executable sections into ELF binaries, update headers appropriately, and have injected code call functions from the original binary.

Pre-Test Setup: Create a test binary that calls external functions, and a more complex hook that itself calls original functions:

```c
// test_m3.c - Test code injection and cross-references

#include <stdio.h>

#include <string.h>


void print_message(const char* msg) {

    printf("Message: %s\n", msg);

}


int main() {

    print_message("Hello from original binary");

    return 0;

}
```

Create injected code that calls `print_message` and adds new functionality:

```c
// inject_m3.c - Code to inject, calling original functions

#include <stdint.h>


// Function prototype for original function

typedef void (*print_message_t)(const char*);

extern print_message_t original_print_message;


void injected_function() {

    // Call original function

    if (original_print_message) {

        original_print_message("Calling from injected code!");

    }



    // New functionality

    const char* extra = "Additional behavior from injected section";

    // We'll need to call printf or implement our own output

}


// We'll need to resolve original_print_message address during injection
```

Compile as position-independent code:

```bash
gcc -g -O0 -fPIC -c inject_m3.c -o inject_m3.o
```

Test Procedure:

1. **Inject New Section**: Run ELFlex with section injection:

```bash
./elflex inject --binary test_m3 --code inject_m3.o --section-name .inject --output test_m3_injected
```

2. **Verify Section Addition**:
   - `readelf -S test_m3_injected` should show `.inject` section with `AX` flags (alloc+execute)
   - `readelf -l test_m3_injected` should show new `PT_LOAD` segment or updated segment sizes
3. **Verify Symbol Resolution**: Check that symbols are resolved:
   - `readelf -s test_m3_injected` should show symbols from injected code
   - Use `objdump -d test_m3_injected` to verify that calls to original functions have correct addresses
4. **Test Execution**: Modify `test_m3.c` to call the injected function (or create a trampoline that redirects to it):

```bash
./test_m3_injected
```

Should produce:

```
Message: Hello from original binary
Message: Calling from injected code!
```

5. **Test Position Independence**: Inject the same code into different binaries at different addresses to verify PIC works.
6. **Test Large Injection**: Inject code larger than page size to test segment expansion.

Expected Output Matrix:

Test Case	Expected Result	Verification Method
Inject small code section	New `.inject` section appears, binary runs	`readelf` shows section, execution succeeds
Inject code calling original function	Cross-reference resolved, call works	Injected code successfully calls original function
Inject into binary without symbol table	Returns `ERR_SYMBOL_NOT_FOUND` for unresolved symbols	Tool returns error code 5 or applies relocation fixups
Inject code requiring relocation	Relocations applied correctly	No segmentation faults, correct addresses in disassembly
Multiple injections	All sections added, headers updated	`readelf` shows multiple injected sections

Success Criteria:

- ✅ Injected section appears in ELF with correct permissions
- ✅ Binary executes with injected code
- ✅ Injected code can call original functions
- ✅ Position-independent code works at different load addresses
- ✅ ELF headers remain valid (passes `readelf` validation)
- ✅ Original binary sections remain functional

Milestone 4 Checkpoint: Runtime Patching via ptrace

What to Test: The ability to attach to a running process, modify its memory to inject trampolines, and detach leaving hooks active.

Pre-Test Setup: Create a long-running test process with observable periodic behavior:

```c
// test_m4.c - Long-running process for runtime patching

#include <stdio.h>

#include <unistd.h>

#include <signal.h>


volatile int keep_running = 1;


void signal_handler(int sig) {

    keep_running = 0;

}


void monitored_function(int iteration) {

    printf("Iteration %d: Normal operation\n", iteration);

    fflush(stdout);

}


int main() {

    signal(SIGINT, signal_handler);

    signal(SIGTERM, signal_handler);



    int count = 0;

    while (keep_running && count < 100) {

        monitored_function(count);

        sleep(1);

        count++;

    }



    printf("Process exiting\n");

    return 0;

}
```

Compile and run in background:

```bash
gcc -g -O0 -no-pie -o test_m4 test_m4.c

./test_m4 &

TEST_PID=$!

echo "Test process PID: $TEST_PID"
```

Create a runtime hook:

```c
// runtime_hook.c - Hook for runtime injection

#include <stdio.h>


void runtime_hook(int iteration) {

    printf("RUNTIME HOOK: Intercepted iteration %d\n", iteration);

    fflush(stdout);

}
```

Compile as position-independent:

```bash
gcc -g -O0 -fPIC -c runtime_hook.c -o runtime_hook.o
```

Test Procedure:

1. **Attach to Running Process**:

```bash
   ./elflex runtime --pid $TEST_PID --function monitored_function --hook runtime_hook.o
```

2. **Verify Process Continues**: The process should continue running after a brief pause (while patching occurs).
3. **Observe Hook Execution**: The process output should now show:

```
   Iteration 0: Normal operation
   RUNTIME HOOK: Intercepted iteration 1
   Iteration 1: Normal operation
   RUNTIME HOOK: Intercepted iteration 2
   Iteration 2: Normal operation
```

4. **Test Memory Protection Handling**: Verify the tool handles read-only code pages by checking `strace` output for `mprotect` calls.
5. **Test Detach and Cleanup**: Send `SIGINT` to test process and verify it exits cleanly:

```bash
   kill -INT $TEST_PID

   wait $TEST_PID

   echo "Exit status: $?"
```

6. **Test Error Conditions**: Attempt to attach to non-existent PID, process without permissions, etc.

Expected Output Matrix:

Test Case	Expected Result	Verification Method
Attach to running process	Process pauses briefly, then continues with hook	Output shows hook messages interleaved
Attach to non-existent PID	Returns `ERR_PTRACE_PERM` or similar	Tool returns error code 100
Process without ptrace permissions (YAMA)	Returns `ERR_PTRACE_PERM`	Tool returns error code 100 with descriptive message
Hook on function in shared library	Trampoline injected into library code segment	Hook fires for library calls
Multiple runtime hooks	All hooks installed correctly	Multiple hook messages appear
Detach after patching	Process continues running independently	Process runs after ELFlex exits

Success Criteria:

- ✅ Process continues running after patching (doesn't crash)
- ✅ Hook executes on each function call

- ✅ Original function behavior preserved
- ✅ Process can be cleanly terminated
- ✅ Memory protection changed and restored correctly
- ✅ Instruction cache flushed (no stale code execution)
- ✅ Error conditions handled gracefully without leaving process stuck

Implementation Guidance

The testing strategy described above requires specific infrastructure to implement effectively. This guidance provides concrete code and structure for building the test suite.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Test Runner	Shell scripts with `bash`	Python `pytest` with custom plugins
Binary Comparison	`diff` and `cmp` commands	Custom binary diff with granular reporting
Test Binary Generation	Manual compilation with `gcc`	CMake/`make` based test suite with dependency tracking
Memory Checking	Manual `valgrind` runs	Integrated ASan builds with `-fsanitize=address`
Coverage Analysis	`gcov` with `gcc`	`llvm-cov` with Clang instrumentation

B. Recommended File/Module Structure

```
elflex-project/
├── src/                          # Main tool source code
│   ├── elf_loader.c
│   ├── patch_planner.c
│   ├── trampoline_gen.c
│   ├── code_injector.c
│   └── runtime_patcher.c
├── tests/                        # Test suite
│   ├── unit/                     # Unit tests
│   │   ├── test_elf_loader.c
│   │   ├── test_patch_planner.c
│   │   ├── test_trampoline_gen.c
│   │   └── test_utils.c
│   ├── integration/              # Integration tests
│   │   ├── test_binaries/        # Source for test binaries
│   │   │   ├── m1_simple.c
│   │   │   ├── m2_hookable.c
│   │   │   ├── m3_injectable.c
│   │   │   └── m4_longrun.c
│   │   ├── hooks/                # Hook functions for testing
│   │   │   ├── hook_m2.c
│   │   │   └── runtime_hook.c
│   │   └── test_integration.sh   # Integration test runner
│   ├── fixtures/                 # Pre-built test binaries
│   │   ├── simple.elf
│   │   └── libc_dependent.elf
│   └── utils/                    # Testing utilities
│       ├── test_runner.c
│       ├── binary_comparator.c
│       └── memory_check.c
├── scripts/                      # Build and test scripts
│   ├── build_tests.sh
│   ├── run_tests.sh
│   └── clean_tests.sh
└── Makefile                      # Build system
```

C. Infrastructure Starter Code (Complete Test Runner)

```c
/* tests/utils/test_runner.c - Complete test runner infrastructure */

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/stat.h>


#define MAX_TEST_NAME 256

#define MAX_COMMAND 1024


typedef struct {

    char name[MAX_TEST_NAME];

    char command[MAX_COMMAND];

    int expected_exit_code;

    char output_file[256];

    char expected_output_file[256];

} test_case_t;


typedef struct {

    test_case_t* tests;

    int count;

    int passed;

    int failed;

} test_suite_t;


/* Create a new test suite */

test_suite_t* test_suite_create() {

    test_suite_t* suite = malloc(sizeof(test_suite_t));

    if (!suite) return NULL;



    suite->tests = NULL;

    suite->count = 0;

    suite->passed = 0;

    suite->failed = 0;
```

```
    return suite;
}

/* Add a test case to the suite */
int test_suite_add(test_suite_t* suite, const char* name, const char* command,
                   int expected_exit, const char* output_file,
                   const char* expected_output) {
    if (!suite || !name || !command) return 0;

    test_case_t* new_tests = realloc(suite->tests,
                                     (suite->count + 1) * sizeof(test_case_t));
    if (!new_tests) return 0;

    suite->tests = new_tests;
    test_case_t* test = &suite->tests[suite->count];

    strncpy(test->name, name, MAX_TEST_NAME - 1);
    test->name[MAX_TEST_NAME - 1] = '\0';

    strncpy(test->command, command, MAX_COMMAND - 1);
    test->command[MAX_COMMAND - 1] = '\0';

    test->expected_exit_code = expected_exit;

    if (output_file) {
        strncpy(test->output_file, output_file, sizeof(test->output_file) - 1);
        test->output_file[sizeof(test->output_file) - 1] = '\0';
    } else {
        test->output_file[0] = '\0';
    }

    if (expected_output) {
        strncpy(test->expected_output_file, expected_output,
                sizeof(test->expected_output_file) - 1);
        test->expected_output_file[sizeof(test->expected_output_file) - 1] = '\0';
    } else {
```

```
        test->expected_output_file[0] = '\0';
    }

    suite->count++;
    return 1;
}

/* Run a single test case */
int run_test_case(const test_case_t* test) {
    printf("[RUN] %s\n", test->name);

    // Execute the command
    int exit_code = system(test->command);
    int actual_exit = WEXITSTATUS(exit_code);

    // Check exit code
    if (actual_exit != test->expected_exit_code) {
        printf("[FAIL] %s: Expected exit %d, got %d\n",
               test->name, test->expected_exit_code, actual_exit);
        return 0;
    }

    // If output file specified, compare with expected
    if (test->output_file[0] && test->expected_output_file[0]) {
        char compare_cmd[512];
        snprintf(compare_cmd, sizeof(compare_cmd),
                 "diff -q %s %s > /dev/null 2>&1",
                 test->output_file, test->expected_output_file);

        int diff_result = system(compare_cmd);
        if (diff_result != 0) {
            printf("[FAIL] %s: Output differs from expected\n", test->name);
            return 0;
        }
    }
```

```
    printf("[PASS] %s\n", test->name);
    return 1;
}

/* Run all tests in the suite */
void test_suite_run(test_suite_t* suite) {
    if (!suite) return;

    printf("=== Running Test Suite: %d tests ===\n", suite->count);

    for (int i = 0; i < suite->count; i++) {
        if (run_test_case(&suite->tests[i])) {
            suite->passed++;
        } else {
            suite->failed++;
        }
    }

    printf("\n=== Results: %d passed, %d failed ===\n",
           suite->passed, suite->failed);
}

/* Clean up test suite */
void test_suite_free(test_suite_t* suite) {
    if (!suite) return;

    free(suite->tests);
    free(suite);
}
```

D. Core Test Skeleton Code

```c
/* tests/unit/test_elf_loader.c - Unit test skeleton for ELF Loader */

#include "../utils/test_runner.h"

#include "../../src/elf_loader.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>


/* TODO 1: Test valid ELF file loading */

void test_load_valid_elf() {

    // TODO 1.1: Create or copy a known valid ELF file to test directory

    // TODO 1.2: Call elf_load() on the file

    // TODO 1.3: Verify returned elf_binary_t is not NULL

    // TODO 1.4: Verify ehdr field points to valid ELF header

    // TODO 1.5: Verify section_count > 0

    // TODO 1.6: Call elf_binary_free() to clean up

    // TODO 1.7: Assert no memory leaks (use valgrind in actual test)

}


/* TODO 2: Test invalid ELF file handling */

void test_load_invalid_elf() {

    // TODO 2.1: Create a file with garbage data (not ELF format)

    // TODO 2.2: Call elf_load() on the file

    // TODO 2.3: Verify function returns NULL or sets error state

    // TODO 2.4: Verify appropriate error code is set (ERR_INVALID_ELF)

}


/* TODO 3: Test section finding */

void test_find_section() {

    // TODO 3.1: Load a test ELF with known sections (.text, .data)

    // TODO 3.2: Call elf_find_section(binary, ".text")

    // TODO 3.3: Verify returned section_t is not NULL

    // TODO 3.4: Verify section->shdr->sh_type == SHT_PROGBITS

    // TODO 3.5: Verify section->shdr->sh_flags includes SHF_EXECINSTR

    // TODO 3.6: Test with non-existent section name (should return NULL)

}
```

```c
/* TODO 4: Test symbol resolution */
void test_get_symbol_address() {
    // TODO 4.1: Load a test ELF with known symbols (e.g., 'main')
    // TODO 4.2: Call elf_get_symbol_address(binary, "main")
    // TODO 4.3: Verify returned address is within .text section
    // TODO 4.4: Test with non-existent symbol (should return 0 or error)
    // TODO 4.5: Test with stripped binary (should return 0 or error)
}

/* TODO 5: Test ELF integrity validation */
void test_is_valid_elf() {
    // TODO 5.1: Create buffer with valid ELF magic bytes
    // TODO 5.2: Call is_valid_elf() on buffer
    // TODO 5.3: Verify returns 1 (true)
    // TODO 5.4: Test with corrupted magic bytes (should return 0)
    // TODO 5.5: Test with partial buffer (size < EI_NIDENT)
}

/* Main test runner */
int main() {
    test_suite_t* suite = test_suite_create();

    // TODO: Add test cases using test_suite_add()
    // test_suite_add(suite, "Valid ELF load", "./run_single_test test_load_valid_elf",
    //                0, "output.log", "expected_valid.log");

    test_suite_run(suite);

    int result = (suite->failed == 0) ? 0 : 1;
    test_suite_free(suite);

    return result;
}
```

E. Language-Specific Hints (C)

1. **Memory Testing**: Always compile test binaries with `-g` for debug symbols and `-fsanitize=address` for AddressSanitizer during development:

```bash
gcc -g -fsanitize=address -o test_program test_program.c
```

2. **Test Isolation**: Use temporary directories for each test to avoid interference:

```c
char tmpdir[] = "/tmp/elflex_test_XXXXXX";

mkdtemp(tmpdir);

chdir(tmpdir);

// Run test

chdir("..");

rmdir(tmpdir);
```

3. **Process Testing**: For runtime patching tests, use `fork()` to create child processes:

```c
pid_t child = fork();

if (child == 0) {

    // Child process: run test binary

    execl("./test_m4", "test_m4", NULL);

} else {

    // Parent process: run ELFlex on child PID

    sleep(1); // Let child start

    char cmd[256];

    snprintf(cmd, sizeof(cmd), "./elflex runtime --pid %d", child);

    system(cmd);

    waitpid(child, NULL, 0);

}
```

4. **Binary Comparison**: Use `memcmp()` for precise binary comparison but include hex dump on mismatch:

```c
if (memcmp(original, patched, size) != 0) {

    printf("Binary mismatch at offset %ld\n",

           first_diff_offset(original, patched, size));

    hex_dump(original, size, 0);

    hex_dump(patched, size, 0);

}
```

F. Milestone Checkpoint Verification Script

```bash
#!/bin/bash

# scripts/run_milestone_check.sh

# Run comprehensive tests for a specific milestone

MILESTONE=$1

echo "=== Testing Milestone $MILESTONE ==="

case $MILESTONE in

    1)

        echo "Testing Instruction-level Patching"

        # Build test binary

        gcc -g -O0 -no-pie -o tests/fixtures/m1_test tests/integration/test_binaries/m1_simple.c



        # Run unit tests for patch planner

        ./tests/unit/test_patch_planner



        # Run integration test

        ./tests/integration/test_m1_patching.sh



        # Verify with objdump

        objdump -d tests/fixtures/m1_test_patched | grep -A5 -B5 "target_function"

        ;;

    2)

        echo "Testing Function Hooking with Trampolines"

        # Build test binary and hook

        gcc -g -O0 -no-pie -o tests/fixtures/m2_test tests/integration/test_binaries/m2_hookable.c

        gcc -g -O0 -fPIC -c tests/integration/hooks/hook_m2.c -o tests/fixtures/hook_m2.o



        # Run trampoline generator tests

        ./tests/unit/test_trampoline_gen



        # Run integration test

        ./tests/integration/test_m2_hooking.sh



        # Verify execution
```

```
        ./tests/fixtures/m2_test_hooked 2>&1 | grep -q "HOOK:"
        if [ $? -eq 0 ]; then
            echo "Hook executed successfully"
        else
            echo "ERROR: Hook not executed"
            exit 1
        fi
        ;;
    3)
        echo "Testing Code Injection"
        # Similar pattern for milestone 3
        ;;
    4)
        echo "Testing Runtime Patching"
        # Note: Requires ptrace permissions, may need sudo or special setup
        # Consider running in Docker container for isolation
        ;;
    *)
        echo "Usage: $0 [1|2|3|4]"
        exit 1
        ;;
esac

echo "=== Milestone $MILESTONE tests completed ==="
```

G. Debugging Tips Table

Symptom	Likely Cause	How to Diagnose	Fix
Segmentation fault after patching	Modified instruction corrupted control flow or stack	1. Use `gdb` to catch segfault 2. Examine `bt` (backtrace) 3. Disassemble patched area with `objdump` 4. Check for corrupted return addresses	Ensure patch doesn't cross instruction boundaries, maintain stack alignment
Hook not called	Trampoline jump not installed or hook address wrong	1. `objdump -d` to verify prologue replaced with `jmp` 2. Check trampoline code contains correct hook address 3. Single-step with `gdb` to see execution flow	Verify symbol resolution, check RIP-relative addressing in trampoline
Process hangs after ptrace attach	Process stopped but not resumed, deadlock with signals	1. Use `strace -p <pid>` to see blocked syscalls 2. Check for pending signals with `kill -l <pid>` 3. Examine tool's state machine	Ensure `PTRACE_CONT` is called after modifications, handle signals properly
Modified binary fails to load	ELF headers corrupted during injection	1. Run `readelf -h` on patched binary 2. Check section offsets alignment 3. Verify no segment overlap	Recalculate all offsets after injection, maintain alignment constraints
Memory leak in tool	Resources not freed on error paths	1. Run with `valgrind --leak-check=full` 2. Check all `malloc` have matching `free` 3. Look for early returns without cleanup	Implement consistent ownership hierarchy, use `goto cleanup` pattern
Runtime patch works once then fails	Instruction cache not flushed	1. Check if `__builtin___clear_cache` is called 2. Verify on architecture needing explicit cache flush	Add cache flush after `PTRACE_POKEDATA` on architectures requiring it
Hook causes infinite recursion	Hook calls hooked function without trampoline bypass	1. Trace call stack in `gdb` 2. Check hook doesn't directly call original symbol	Use function pointer to original code, not symbol name

Debugging Guide

> **Milestone(s):** 1 (Instruction-level Patching), 2 (Function Hooking with Trampolines), 3 (Code Injection), 4 (Runtime Patching via ptrace)

Binary instrumentation is inherently delicate work—modifying a compiled binary is akin to performing live surgery on a patient who cannot tell you what hurts. The most frustrating bugs often manifest not in the instrumentation tool itself, but in the patched binary's behavior or in the target process during runtime injection. This guide provides a systematic approach to diagnosing and fixing common problems that arise across all four milestones.

Think of debugging instrumentation bugs as being a **forensic investigator** and a **mechanic** simultaneously. You must first examine the crime scene (the corrupted binary or crashed process) to gather clues, then apply precise mechanical fixes to the underlying cause. The symptoms are often dramatic (crashes, hangs, incorrect output), but the root causes are usually subtle mistakes in one of the five core components described in the architecture.

Symptom → Cause → Diagnosis → Fix Table

The following table organizes common symptoms by their most likely causes, provides diagnostic steps to confirm the hypothesis, and suggests specific fixes. Symptoms are grouped by the milestone where they typically appear.

Symptom	Likely Cause	Diagnosis Steps	Fix
Segmentation fault immediately when running patched binary (Milestone 1)	1. **Instruction patching at non-instruction boundary**: Replacing bytes in middle of multi-byte instruction. 2. **Corrupted ELF headers**: Invalid section offsets or sizes after patching. 3. **Overwritten critical data**: Accidentally patched data in `.rodata` or `.data` sections thinking it was code.	1. Use `objdump -d original_binary` and `objdump -d patched_binary` to compare disassembly at patch offset. Check if instruction boundaries align. 2. Run `readelf -h patched_binary` and `readelf -l patched_binary` to verify program headers point to valid offsets. 3. Check `readelf -S patched_binary` to confirm patch offset falls within `.text` section (look at `sh_addr` and `sh_size`).	1. Ensure patch offsets are aligned to instruction boundaries. Use disassembler library or manual verification. 2. Validate that `binary_write` correctly updates all affected offsets when sections grow or shift. 3. Add boundary check: verify offset is within executable section (`SHF_EXECINSTR` flag).
Patched binary runs but behaves incorrectly (wrong output, infinite loop) (Milestone 1)	1. **Relative branch target miscalculation**: Patched instruction contains relative jump/call whose offset wasn't adjusted for new instruction size. 2. **Accidental NOP sled overflow**: NOP padding extends into next instruction, corrupting it. 3. **Endianness mismatch**: Writing multi-byte instructions with wrong byte order for target architecture.	1. Disassemble patched region and manually compute jump targets. Look for jumps that now point into middle of instructions. 2. Examine disassembly around patch: look for unexpected NOP instructions after intended padding region. 3. Check `e_ident[EI_DATA]` in ELF header (1=little, 2=big). Verify instruction encoding matches architecture endianness.	1. When patching relative branches, recalculate offset based on new instruction location. Better yet, avoid patching relative branches initially. 2. Ensure `nop_padding_size` doesn't exceed space until next instruction boundary. 3. Use architecture-specific instruction encoding functions that respect host/target endianness.
Hook function never called (Milestone 2)	1. **Incorrect symbol resolution**: Hook address points to wrong location (maybe PLT stub vs. actual function). 2. **Trampoline jump not installed**: Prologue replacement failed or wrote wrong bytes. 3. **Trampoline code itself crashes**: Register corruption or RIP-relative addressing error in relocated prologue.	1. Compare `elf_get_symbol_address` result with `objdump -t binary | grep function`. Check for symbols with `@plt` suffix. 2. Use `objdump -d` on patched binary at function entry; should see jump instruction (e.g., `jmp 0x...`). If not, check `apply_patch` success. 3. Single-step through trampoline in debugger (`gdb -q patched_binary`, break at trampoline entry). Watch for segfault during relocated prologue.	1. For PLT functions, consider hooking the PLT entry itself or resolve to actual function via dynamic linking info. Document this limitation. 2. Verify jump offset calculation: `target_address - (patch_address + jump_instruction_size)`. For x86-64 relative jump, ensure offset fits in 32 bits. 3. Implement `fixup_rip_relative` for relocated instructions that use RIP-relative addressing (common in x86-64).
Hook called once then crash on return (Milestone 2)	1. **Stack imbalance**: Trampoline modifies stack pointer without restoring it. 2. **Register corruption**: Trampoline clobbers a caller-saved register that original function expects preserved. 3. **Wrong calling convention**: Hook function uses different calling convention (e.g., `stdcall` vs `cdecl`) than target.	1. Examine trampoline assembly: count `push` / `pop` instructions; they must balance. Use `gdb` to print `rsp` before/after trampoline. 2. Check trampoline's register save/restore sequence. All registers (except those explicitly allowed to be clobbered) must be saved before hook call and restored after. 3. Verify hook function signature matches target (parameters, return type). On x86-64, check for proper use of `rax`, `rdi`, `rsi`, etc.	1. Ensure trampoline saves `rsp` if it modifies it (e.g., via `sub rsp, ...` for alignment). 2. Save all general-purpose registers (or at least those not used for parameter passing) to stack. Use `pusha`/`popa` if available, or explicit push/pop sequence. 3. Make hook function use standard System V AMD64 ABI (if Linux x86-64). Use `extern "C"` to prevent C++ name mangling.
Injected code section not loaded at expected	1. **Segment overlap**: New `PT_LOAD` segment overlaps existing segment's virtual address range.	1. `readelf -l patched_binary` shows all segments; check for overlapping `p_vaddr` ranges. 2. Check `sh_addralign` vs segment's	1. Choose injection address that doesn't overlap existing segments. Common approach: place after last `PT_LOAD` segment at next page

Symptom	Likely Cause	Diagnosis Steps	Fix
address (Milestone 3)	2. **Alignment mismatch**: Injected section's `sh_addr` not aligned to `p_align` of its containing segment. 3. **Missing program header**: Injected section not covered by any `PT_LOAD` segment, so loader ignores it.	`p_align`. Typically need page alignment (0x1000). 3. Verify injected section's `sh_addr` falls within some `PT_LOAD` segment's `p_vaddr` to `p_vaddr + p_memsz`.	boundary. 2. Align `sh_addr` to maximum of `sh_addralign` and segment's `p_align`. 3. Either add new `PT_LOAD` header with `inject_section` or expand existing executable segment to cover new section.
Injected code crashes when calling original function (Milestone 3)	1. **Absolute address not patched**: Injected code contains hardcoded address that's invalid in new location. 2. **Position-dependent code injected**: Code expects to be at specific address but was loaded elsewhere. 3. **Symbol resolution failure**: `resolve_symbol_in_target` returns wrong address (maybe local vs global symbol).	1. Disassemble injected code: look for `call 0x...` or `mov rax, 0x...` with addresses from original binary. 2. Check if injected code was compiled as position-independent (`-fPIC`). Use `is_likely_pic` on injected bytes. 3. Compare resolved address with `nm -D original_binary | grep function`. Ensure looking at dynamic symbols if needed.	1. Use `patch_absolute_address` to fix up absolute references in injected code after determining load address. 2. Always compile hook code with `-fPIC -pie` to generate position-independent code. 3. Search both `.symtab` and `.dynsym` sections for symbol. Prefer `.dynsym` for functions that may be interposed.
ptrace attach fails with permission denied (Milestone 4)	1. **YAMA ptrace_scope restriction**: Linux security module blocks non-child ptrace. 2. **Process running as different user**: Tool lacks privileges to trace target process. 3. **Process already being traced**: Another debugger (or anti-debug) holds ptrace lock.	1. Check `cat /proc/sys/kernel/yama/ptrace_scope`. Values: 0=classic, 1=restricted, 2=admin-only, 3=no attach. 2. Verify tool runs as same user or as root. Use `ps -p <pid> -o user`. 3. Check `ls -l /proc/<pid>/status` for `TracerPid:` field (non-zero means traced).	1. Run as root, or adjust YAMA: `sudo sysctl kernel.yama.ptrace_scope=0` (temporary). Document security implications. 2. Use `sudo` or setcap: `sudo setcap cap_sys_ptrace=eip tool`. 3. If anti-debug, may need to detach other tracer first (complex). For testing, use simple binaries without protection.
Process hangs after ptrace attach (Milestone 4)	1. **Signal mishandling**: Process stopped but signals not forwarded correctly. 2. **Deadlock in multi-threaded process**: One thread stopped while others continue, waiting on locks. 3. **Infinite loop in trampoline**: Bad code causes infinite loop in injected trampoline.	1. Use `strace -p <pid>` to see if process is making syscalls (shouldn't if stopped). Check for pending signals with `waitpid`. 2. Check if other threads are running (`ps -T -p <pid>`). If tool only stops one thread, others may deadlock. 3. Attach debugger to hanged process (another terminal) and examine instruction pointer.	1. Implement proper signal forwarding: after `PTRACE_CONT`, intercept and forward signals from tracer to tracee. 2. Stop all threads using `PTRACE_ATTACH` on each thread ID. Complex; for milestone 4, assume single-threaded targets. 3. Debug trampoline code offline first. Use `PTRACE_PEEKTEXT` to verify written bytes match expected trampoline.
Hook works but original function behavior broken (Milestone 4)	1. **Instruction cache not flushed**: CPU executes stale cached instructions instead of new trampoline. 2. **Memory protection not restored**: Page left writable, causing segmentation fault on execution. 3. **Race condition**: Another thread	1. Check architecture: x86 has coherent I-cache, but ARM/PPC require explicit flush. Still good practice to call `flush_instruction_cache`. 2. Use `gdb` to examine `/proc/<pid>/maps` after patching; code pages should be `r-x`, not `rwx`. 3. Hard to diagnose; add logging to see if patch is applied multiple times.	1. Always call `flush_instruction_cache` (or `__builtin___clear_cache` on Linux) after code modification. 2. In `make_page_writable`, save original protections and restore them after write. 3. For multi-threaded, stop all threads during patch. For milestone 4, document this limitation.

Symptom	Likely Cause	Diagnosis Steps	Fix
	executed original code while patch was being written.		
Process crashes after detach (Milestone 4)	1. **Corrupted register state**: Ptrace changed registers during stop and didn't restore them. 2. **Partial patch written**: Only part of trampoline written before detach (e.g., due to error). 3. **Stack corruption**: Trampoline or hook corrupted stack, but crash deferred until after detach.	1. Save original registers with `PTRACE_GETREGS` before any modification and restore with `PTRACE_SETREGS` before detach. 2. Verify written memory with `PTRACE_PEEKTEXT` after write. Checksum trampoline bytes. 3. Examine core dump (if enabled) with `gdb -c core.<pid>` to see stack trace.	1. `runtime_session_t` should store `original_regs` and restore them in `detach_from_process`. 2. Write entire trampoline atomically if possible (small). For larger patches, verify after write. 3. Ensure trampoline preserves stack alignment (16-byte boundary) and doesn't overflow buffer.

Debugging Tools and Techniques

Effective debugging of binary instrumentation requires a specialized toolkit. Beyond traditional debuggers, you'll need tools that inspect binary structures, trace system calls, and examine low-level process state. This section describes both external tools and internal instrumentation you should build into ELFlex itself.

External Tools: The Forensic Toolkit

> Think of these tools as the **investigator's crime lab equipment**. Each provides a different lens through which to examine the binary or process, revealing specific types of evidence about what went wrong.

Tool	Primary Use	Key Commands & Interpretation
`readelf`	Inspecting ELF structure and integrity.	`readelf -h binary` – Header sanity check (entry point, program header offset). `readelf -l binary` – Program headers (segments) and their memory layout. Critical for detecting overlap. `readelf -S binary` – Section headers and their offsets. Verify `.text` and `.inject` sections exist. `readelf -s binary` – Symbol table. Check if target function symbol exists and has correct address. `readelf -r binary` – Relocations. Important for injected code that references original symbols.
`objdump`	Disassembling and comparing code before/after patching.	`objdump -d binary` – Full disassembly. Pipe to `less` or redirect to file for comparison. `objdump -d --start-address=0x... --stop-address=0x... binary` – Disassemble specific range around patch site. `objdump -t binary` – Symbol table in simpler format than `readelf -s`. `objdump -h binary` – Section headers (similar to `readelf -S`). **Pro tip**: `diff -u original.disasm patched.disasm` to spot unintended changes.
`gdb`	Dynamic analysis of patched binaries and runtime processes.	`gdb -q ./patched_binary` – Start debugger on patched binary. `break *0xaddress` – Set breakpoint at absolute address (e.g., trampoline entry). `x/i 0xaddress` – Examine instruction at address. `stepi` – Single instruction step. Crucial for tracing trampoline execution. `info registers` – Inspect all register values before/after hook. `gdb -p <pid>` – Attach to running process (alternative to ptrace for debugging). `generate-core-file` – Create core dump of running process for post-mortem analysis.
`strace`	Understanding system call interactions, especially ptrace.	`strace -f -o trace.txt ./patcher args` – Trace tool with child processes (`-f`). `strace -p <pid>` – Attach to running process to see syscalls (if not ptrace-stopped). **Key signals**: Look for `PTRACE_` calls and their arguments/return values. `EACCES` indicates permission issues. `ESRCH` means process doesn't exist.
`nm`	Quick symbol lookup.	`nm -D binary` – Dynamic symbols (for shared library functions). `nm binary` – Regular symbols. Check if target is defined (`T`) or undefined (`U`). `nm -n binary` – Sort by address, helpful for finding nearest symbol to an address.
`hexdump` / `xxd`	Raw byte inspection.	`xxd -g 1 -s offset -l 32 binary` – View 32 bytes at offset. Compare original vs patched files. `diff -u <(xxd original) <(xxd patched)` – Visualize all byte differences.
`/proc/<pid>/maps`	Process memory layout at runtime.	`cat /proc/<pid>/maps` – Shows memory regions, permissions, and mapped files. Verify injected code appears with `r-xp` permissions. **Key insight**: If your injected section doesn't appear, it wasn't loaded (missing `PT_LOAD` segment).
`/proc/<pid>/mem`	Direct memory access (alternative to ptrace).	While less atomic than ptrace, useful for manual inspection: `dd if=/proc/<pid>/mem bs=1 skip=$((0xaddr)) count=16 2>/dev/null | xxd`

Internal Instrumentation: Building Diagnostics into ELFlex

> The most effective debugging comes from **instrumenting the instrumenter**. By building comprehensive logging, validation, and diagnostic modes into ELFlex itself, you can see exactly what decisions the tool makes and where it goes wrong.

1. Structured Logging System

Instead of scattered `printf` statements, implement a configurable logging system with levels (DEBUG, INFO, WARN, ERROR) that can be enabled at runtime. Key places to add logging:

Component	What to Log	Diagnostic Value
`elf_load`	Section addresses, sizes, and flags.	Confirms correct parsing of target binary.
`plan_patch`	Patch offset, original bytes, new bytes, padding size.	Documents exactly what changes are planned.
`apply_patch`	Success/failure, actual bytes written.	Verifies patch application.
`generate_trampoline`	Trampoline code bytes in hex, hook address, target address.	Allows disassembly of generated trampoline.
`inject_section`	Injected section virtual address, size, and containing segment.	Confirms injection location matches plan.
`attach_to_process`	Ptrace return values, process state.	Shows attachment success and any errors.
`write_process_memory`	Address, bytes written, verification checksum.	Catches memory write failures.

2. Validation Mode

Implement a `--validate` flag that performs extensive checks without actually modifying anything:

- Verify all patch offsets are within executable sections.
- Check that trampoline jumps are within range (±2GB for relative jumps).
- Ensure injected section address doesn't overlap existing segments.
- Confirm symbol resolutions produce valid addresses.
- Simulate ptrace operations (check permissions, process existence).

3. Dry-run and Diff Generation

For static patching (Milestones 1-3), implement `--dry-run` that:

1. Loads the binary and plans all modifications.
2. Creates a temporary copy with patches applied.
3. Generates a detailed report of changes (bytes changed, sections added).
4. Produces unified diff of disassembly (`objdump -d` output) between original and patched.
5. Cleans up temporary files without saving.

4. Built-in Disassembler Integration

While the project prerequisites include a disassembler, integrate it directly for debugging:

- Before applying any patch, disassemble 16 bytes before and after the target offset.
- Log the instructions to confirm alignment.
- After patch generation, disassemble the trampoline code to verify it's valid.
- When RIP-relative fixups are applied, disassemble before/after to confirm correction.

5. State Dumping for Runtime Patching

For Milestone 4, implement commands to dump process state:

- `--dump-registers <pid>` : Show current register state via `PTRACE_GETREGS` .
- `--dump-memory <pid> <addr> <size>` : Hex dump of process memory.
- `--find-trampoline <pid>` : Scan process memory for known trampoline patterns.

Diagnostic Workflow: A Step-by-Step Approach

When faced with a bug, follow this systematic workflow:

Step 1: Reproduce and Isolate

- Can you reproduce the bug consistently?

- Does it happen with a simple test binary (write a minimal C program that calls a target function)?
- Does it happen only with specific binaries (e.g., stripped vs. not, PIE vs. non-PIE)?

Step 2: Enable Maximum Logging

- Run ELFlex with `--log-level=DEBUG --log-file=debug.log`.
- Examine the log for the first error or unexpected decision.

Step 3: Examine the Evidence

- For static patching bugs: Use `readelf` and `objdump` on both original and patched binaries.
- For runtime patching bugs: Use `gdb` to attach to the target process *after* instrumentation (if it's still running) or examine core dumps.

Step 4: Create Minimal Test Case

- Reduce the target binary to smallest possible (e.g., a 10-line C program).
- Reduce the hook to simplest possible (e.g., a function that just returns).
- This eliminates confounding factors.

Step 5: Hypothesis and Experiment

- Based on the symptom table above, form a hypothesis.
- Create an experiment to test it (e.g., "If I comment out the RIP-relative fixup, does the bug disappear?").
- Use debugging tools to gather evidence for/against your hypothesis.

Step 6: Fix and Verify

- Apply the fix.
- Verify with the minimal test case first.
- Then run the full test suite.

Common Pitfalls in Debugging Approach

⚠️ Pitfall: Jumping to conclusions without evidence

- **Why it's wrong:** Many instrumentation bugs have similar symptoms but different root causes. Assuming the cause without evidence leads to wasted time fixing the wrong thing.
- **How to avoid:** Always gather data first. Use the diagnostic tools above to collect evidence before forming a hypothesis.

⚠️ Pitfall: Not checking the obvious

- **Why it's wrong:** Developers often assume complex bugs have complex causes, but many instrumentation failures are due to simple issues: wrong file permissions, incorrect offset calculations, or typos in constant definitions.
- **How to avoid:** Start with basic sanity checks: Did the tool open the file successfully? Is the target process running? Are you compiling with the right architecture flags?

⚠️ Pitfall: Ignoring architecture differences

- **Why it's wrong:** x86-64 has specific behaviors (RIP-relative addressing, red zones) that differ from other architectures. Assuming universal behavior causes subtle bugs.
- **How to avoid:** Document architecture assumptions explicitly. When debugging, consult the architecture manual for instruction encoding and ABI details.

⚠️ Pitfall: Forgetting about ASLR

- **Why it's wrong:** Address Space Layout Randomization changes load addresses between runs. Hardcoded addresses from one run won't work in the next.
- **How to avoid:** For static patching, use offsets relative to section bases or virtual addresses that will be consistent after loading. For runtime patching, always read actual addresses from `/proc/<pid>/maps`.

Implementation Guidance

Building robust debugging capabilities into ELFlex from the start will save countless hours. This section provides concrete implementation patterns.

Technology Recommendations Table

Component	Simple Option	Advanced Option
Logging	Simple `fprintf(stderr, ...)` with `#ifdef DEBUG` conditionals.	`syslog()` for system-wide logging or custom logging library with levels, rotation, and structured output (JSON).
Disassembly	External call to `objdump` via `popen()` and parsing text output.	Integrated disassembler library (Capstone, Zydis) for programmatic inspection.
Hex Dumping	Simple loop with `printf("%02x ", byte)`.	Using `libiberty`'s `hex_dump()` or custom ANSI-colored diff output.
Process Inspection	Reading `/proc/<pid>/` files directly.	Using `libproc` or `ptrace` for more detailed process state.

Recommended File/Module Structure

Add debugging modules to the project structure:

```
elflex/
  src/
    core/                # Core components
    debug/               # Debugging utilities
      log.c              # Logging system implementation
      log.h
      disasm.c           # Disassembler wrapper
      disasm.h
      validate.c         # Validation functions
      validate.h
      hexdump.c          # Hex dump utilities
      hexdump.h
    tools/               # Diagnostic tools
      inspect.c          # CLI tool to inspect binaries
      trace.c            # Runtime tracing tool
  tests/
    debug/               # Debugging test cases
      minimal_binary.c   # Simple test binary
      hook_example.c     # Example hook for testing
```

Infrastructure Starter Code: Logging System

Here's a complete, ready-to-use logging system that can be integrated immediately:

```c
// debug/log.h

#ifndef LOG_H

#define LOG_H


#include <stdio.h>

#include <stdarg.h>

#include <time.h>


typedef enum {

    LOG_DEBUG,

    LOG_INFO,

    LOG_WARN,

    LOG_ERROR,

    LOG_FATAL

} log_level_t;


extern log_level_t current_log_level;

extern FILE* log_stream;


void log_init(FILE* stream, log_level_t level);

void log_message(log_level_t level, const char* file, int line, const char* format, ...);

void log_hexdump(log_level_t level, const char* label, const void* data, size_t size);


#define LOG_DEBUG(...) log_message(LOG_DEBUG, __FILE__, __LINE__, __VA_ARGS__)

#define LOG_INFO(...)  log_message(LOG_INFO, __FILE__, __LINE__, __VA_ARGS__)

#define LOG_WARN(...)  log_message(LOG_WARN, __FILE__, __LINE__, __VA_ARGS__)

#define LOG_ERROR(...) log_message(LOG_ERROR, __FILE__, __LINE__, __VA_ARGS__)

#define LOG_FATAL(...) log_message(LOG_FATAL, __FILE__, __LINE__, __VA_ARGS__)


#define HEXDUMP_DEBUG(label, data, size) log_hexdump(LOG_DEBUG, label, data, size)


#endif // LOG_H
```

```c
// debug/log.c

#include "log.h"

#include <stdlib.h>

#include <string.h>


log_level_t current_log_level = LOG_WARN;

FILE* log_stream = stderr;


void log_init(FILE* stream, log_level_t level) {

    log_stream = stream ? stream : stderr;

    current_log_level = level;

}


static const char* level_strings[] = {

    "DEBUG", "INFO", "WARN", "ERROR", "FATAL"

};


static const char* level_colors[] = {

    "\x1b[36m", "\x1b[32m", "\x1b[33m", "\x1b[31m", "\x1b[35m"

};


void log_message(log_level_t level, const char* file, int line, const char* format, ...) {

    if (level < current_log_level) return;



    time_t t = time(NULL);

    struct tm *lt = localtime(&t);

    char timestamp[20];

    strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", lt);



    // Extract just the filename from the full path

    const char* filename = strrchr(file, '/');

    filename = filename ? filename + 1 : file;



    fprintf(log_stream, "%s%s [%s] %s:%d: ",

            level_colors[level], timestamp, level_strings[level],

            filename, line);



    va_list args;
```

```
    va_start(args, format);
    vfprintf(log_stream, format, args);
    va_end(args);

    fprintf(log_stream, "\x1b[0m\n");
    fflush(log_stream);
}

void log_hexdump(log_level_t level, const char* label, const void* data, size_t size) {
    if (level < current_log_level) return;

    const unsigned char* bytes = (const unsigned char*)data;
    fprintf(log_stream, "%sHex dump: %s (%zu bytes)\n",
            level_colors[level], label, size);

    for (size_t i = 0; i < size; i += 16) {
        fprintf(log_stream, "  %04zx: ", i);

        // Print hex bytes
        for (size_t j = 0; j < 16; j++) {
            if (i + j < size) {
                fprintf(log_stream, "%02x ", bytes[i + j]);
            } else {
                fprintf(log_stream, "   ");
            }
            if (j == 7) fprintf(log_stream, " ");
        }

        fprintf(log_stream, " |");
        // Print ASCII characters
        for (size_t j = 0; j < 16; j++) {
            if (i + j < size) {
                unsigned char c = bytes[i + j];
                fprintf(log_stream, "%c", (c >= 32 && c < 127) ? c : '.');
            } else {
                fprintf(log_stream, " ");
```

```
            }
        }
        fprintf(log_stream, "|\n");
    }

    fprintf(log_stream, "\x1b[0m");
    fflush(log_stream);
}
```

Core Logic Skeleton: Validation Function

Here's a skeleton for a comprehensive validation function that should be called before applying any patch:

```c
// debug/validate.c

#include "validate.h"

#include "../core/elf_loader.h"

#include "../core/patch_planner.h"

#include <string.h>


/**

 * Validates a patch site before application.

 * Returns ERR_SUCCESS if valid, otherwise an error code.

 */

error_t* validate_patch(const elf_binary_t* binary, const patch_site_t* patch) {

    error_t* err = error_create();



    if (!binary || !patch) {

        error_set(err, ERR_INVALID_ARG, "NULL binary or patch", "validate_patch");

        return err;

    }



    // TODO 1: Check that target_offset is within .text section

    // - Get .text section via elf_find_section(binary, ".text")

    // - Verify patch->target_offset is between shdr->sh_addr and shdr->sh_addr + shdr->sh_size

    // - If not, set error: ERR_INVALID_OFFSET, "Patch offset not in executable section"



    // TODO 2: Verify offset is at instruction boundary (optional but recommended)

    // - This requires disassembler integration

    // - Try to disassemble instruction starting at target_offset

    // - If disassembly fails, warn but don't error (LOG_WARN)



    // TODO 3: Check that patch doesn't overlap with existing patches

    // - This requires tracking applied patches in binary (add field to elf_binary_t)

    // - For each existing patch, check if ranges overlap

    // - If overlap, set error: ERR_PATCH_OVERLAP, "Patch overlaps existing modification"



    // TODO 4: Validate new_bytes are valid instructions (if disassembler available)

    // - Disassemble new_bytes to ensure they form valid instructions

    // - LOG_DEBUG the disassembly for debugging
```

```
    // TODO 5: Check that relative branches in new_bytes have valid targets
    // - For each instruction in new_bytes, if it's a relative branch/call
    // - Calculate target address relative to patch location
    // - Ensure target is within .text section

    // TODO 6: If patch->new_size < patch->original_size, nop_padding must be provided
    // - Verify patch->nop_padding != NULL if padding needed
    // - Verify patch->nop_padding_size == patch->original_size - patch->new_size

    error_clear(err); // Clear if all validations pass
    return err;
}

/**
 * Validates a trampoline before injection.
 */
error_t* validate_trampoline(const trampoline_t* trampoline) {
    error_t* err = error_create();

    // TODO 1: Check trampoline->code is not NULL
    // TODO 2: Check trampoline->code_size is reasonable (e.g., < 4096 bytes)
    // TODO 3: Validate hook_address is not zero (unless hook is optional)
    // TODO 4: Validate target_function_address is not zero
    // TODO 5: If relocated_prologue exists, verify it doesn't contain invalid instructions after RIP fixup

    error_clear(err);
    return err;
}
```

Language-Specific Hints for C

- **Memory Inspection**: Use `gdb`'s `x` command with different formats: `x/x` for hex, `x/i` for instructions, `x/s` for strings.
- **Signal Handling with ptrace**: Always check `errno` after ptrace calls. Common errors: `ESRCH` (no such process), `EACCES` (permission denied), `EBUSY` (already traced).
- **Instruction Cache Flushing**: On Linux, use `__builtin___clear_cache((void*)addr, (void*)(addr + size))` which works across architectures.
- **Endianness Helpers**: Use `<endian.h>` if available, or write your own: `uint64_t le64toh(uint64_t little_endian)`.
- **Secure Temporary Files**: Use `mkstemp()` instead of `tmpnam()` for dry-run temporary files.

Milestone Checkpoint Debugging

After implementing each milestone, run these specific debugging checks:

Milestone 1 Checkpoint:

```bash
# Create a test binary
echo 'int main() { return 42; }' > test.c
gcc -o test test.c

# Run with verbose logging
./elflex --log-level=DEBUG --patch-offset=0x1145 --patch-bytes="9090" test test_patched 2> debug.log

# Verify patch
objdump -d test > orig.disasm
objdump -d test_patched > patched.disasm
diff -u orig.disasm patched.disasm | head -20  # Should show exactly 2 bytes changed
```

Milestone 2 Checkpoint:

```bash
# Test with a simple function
echo 'int target() { return 42; } int main() { return target(); }' > test2.c
gcc -o test2 test2.c

# Hook target function
./elflex --hook-function=target --hook-handler=my_hook test2 test2_hooked 2>&1 | grep -i "trampoline"

# Debug the hooked binary
gdb -q ./test2_hooked << 'EOF'
break *target
run
stepi 10  # Step through trampoline
info registers
EOF
```

Milestone 4 Checkpoint:

```bash
# Start test process in background
./test2 &
PID=$!

# Attach and inject
./elflex --runtime --pid=$PID --hook-function=target 2> runtime.log

# Verify process still runs
kill -0 $PID && echo "Process alive" || echo "Process died"

# Check for trampoline in memory
gdb -p $PID -q << 'EOF'
x/10i target
detach
quit
EOF
```

Debugging Tips Quick Reference

When you encounter issues, remember this mental checklist:

1. **Static Patching Issues**: Always compare `readelf` and `objdump` output before/after.
2. **Runtime Issues**: Check `/proc/<pid>/maps` for memory permissions and layout.
3. **Hook Not Called**: Verify the first instruction at the target address is your jump.
4. **Crash in Hook**: Single-step through trampoline in debugger, watch stack pointer.
5. **Permission Denied**: Check YAMA ptrace_scope, run as root, or use `setcap`.
6. **Weird Behavior After Patch**: Instruction cache may need flushing (use `__builtin___clear_cache`).
7. **Process Hangs**: Check for pending signals with `waitpid`, ensure all threads are handled.

By building robust debugging infrastructure early and following systematic diagnosis, you'll transform frustrating binary instrumentation bugs into solvable puzzles.

Future Extensions

> **Milestone(s):** *Building upon all previous milestones*

The **Future Extensions** section explores how ELFlex's architecture can evolve beyond its initial scope. While the core milestones establish a functional binary instrumentation foundation, the system's modular design and clear separation of concerns create a platform for sophisticated enhancements. This section examines potential expansion areas, from advanced hooking techniques to cross-platform support and developer-friendly abstractions.

Potential Extension Areas

The current ELFlex design prioritizes foundational correctness and clear architectural boundaries. This strategic focus enables several natural extension pathways that build upon existing components without requiring major redesigns.

Inline Hooking and Hot-Patching

Mental Model: The Traffic Interceptor at the Intersection

> Imagine traditional trampoline-based hooking as redirecting traffic at an entrance ramp (function prologue). Inline hooking, by contrast, is like installing a traffic light system *within* the highway itself—intercepting vehicles at any point along their journey without requiring them to exit first.

Inline hooking represents a more advanced instrumentation technique where hooks are inserted *within* the function body rather than at its entry point. This enables mid-function behavior monitoring, conditional branching modification, or even runtime behavior alteration based on internal state.

Current Architecture Accommodation:

- The **Patch Planner and Binary Patcher** already handles arbitrary instruction replacement, providing the low-level mechanism
- The **ELF Loader and Parser** understands function boundaries and can identify safe patching locations
- The **Trampoline Generator** can be extended to produce context-preserving mid-function detours

Extension Requirements:

1. **Extended `patch_site_t`** to include execution context preservation requirements:

```c
typedef struct {
    patch_site_t base;           // Existing patch infrastructure
    uint64_t saved_registers;    // Bitmask of registers to save/restore
    uint64_t stack_adjustment;   // Required stack space for context
    int preserve_flags;          // Whether to save/restore CPU flags
} inline_patch_t;
```

2. **Context-Aware Patching Algorithm** that analyzes instruction dependencies to ensure safe mid-function insertion:

```
Algorithm: Safe Inline Hook Insertion
1. Disassemble function starting at target offset
2. Analyze next N instructions for data dependencies and control flow
3. Identify minimum instruction boundary for safe hook placement
4. Calculate register usage and stack state at insertion point
5. Generate context preservation wrapper
6. Plan patch with appropriate NOP padding for wrapper removal
```

3. **State Preservation Trampolines** that save and restore exact execution context:

```
Inline Trampoline Structure:
┌─────────────────────┐
| 1. Save CPU flags   |
| 2. Save registers   |
| 3. Adjust stack     |
| 4. Call hook        |
| 5. Restore stack    |
| 6. Restore registers|
| 7. Restore flags    |
| 8. Original insns   |
| 9. Jump to resume   |
└─────────────────────┘
```

Challenges and Considerations:

- **Instruction Length Variability**: x86-64 instructions have variable sizes, making replacement complex
- **Relative Reference Fixups**: Branch targets within the function must be adjusted after insertion
- **Stack Frame Integrity**: Mid-function hooks must not corrupt local variables or return addresses
- **Performance Overhead**: Context saving/restoring adds more overhead than prologue hooks

> **ADR: Prologue vs. Inline Hooking as Primary Interface**
>
> - **Context**: ELFlex currently uses prologue hooking exclusively. Future versions could support both.
> - **Options Considered**:
>   1. **Prologue-Only**: Keep simple, reliable entry-point hooking
>   2. **Inline-Only**: Maximum flexibility for all instrumentation scenarios
>   3. **Dual-Mode**: Support both with appropriate trade-off awareness
> - **Decision**: Extend to dual-mode architecture
> - **Rationale**: Prologue hooking covers 80% of use cases with minimal complexity, while inline hooking enables advanced scenarios like loop monitoring, conditional breakpoints, and state-dependent behavior modification
> - **Consequences**: Increased code complexity, need for sophisticated disassembly, but enables comprehensive instrumentation toolkit

Extension Option	Pros	Cons	Implementation Complexity
Prologue-Only (Current)	Simple, reliable, minimal overhead	Limited to function entry points	Low
Inline-Only	Maximum flexibility, precise control	Complex, high overhead, error-prone	Very High
Dual-Mode	Best of both worlds, progressive disclosure	More code paths, configuration complexity	Medium-High

Function Exit Hooking and Epilogue Instrumentation

Mental Model: The Customs Inspection on Departure

> If function entry hooks are like passport control when entering a country, exit hooks are like customs inspection when leaving—able to examine what's being taken out (return values) and ensure proper cleanup occurred.

Exit hooking enables post-execution analysis, return value modification, and resource cleanup verification. This requires intercepting the function's return path rather than its entry.

Current Architecture Accommodation:

- The **Trampoline Generator** can be extended to produce epilogue trampolines
- The **Patch Planner** can identify return instructions (`ret` , `retq`) for patching
- The **Code Injector** provides space for exit handler functions

Extension Requirements:

1. **Return Instruction Patching** to redirect to exit trampoline:

2. **Epilogue Trampoline Structure** that preserves return values:

```c
typedef struct {
    trampoline_t base;           // Inherit from entry trampoline
    uint64_t return_value_slot;  // Storage for RAX/MM0/XMM0
    uint64_t stack_cleanup_size; // Bytes to remove for stdcall/fastcall
    int modifies_return_value;   // Whether hook can change return
} exit_trampoline_t;
```

3. **Stack Frame Reconstruction** to provide hook context:

```
Exit Trampoline Execution Flow:
1. Save return address from stack
2. Save return value (RAX/XMM0/etc.)
3. Call exit hook with saved context
4. Optionally modify return value
5. Restore return address to stack
6. Jump to original return address
```

Challenges and Considerations:

- **Multiple Return Points**: Functions may have multiple `ret` instructions
- **Tail Call Optimization**: Some functions end with `jmp` instead of `ret`
- **Calling Convention Variations**: Different conventions handle return values differently
- **Stack Unwinding Compatibility**: Debuggers and exception handlers must still work

Multi-Architecture Support (ARM, AArch64, RISC-V)

Mental Model: The Universal Translator

> Currently ELFlex speaks only "x86-64 dialect." Adding ARM support is like hiring translators for other languages—the core concepts (trampolines, patching) remain, but the implementation details (instruction encodings, register conventions) differ significantly.

Modern systems use diverse CPU architectures, particularly in embedded and mobile domains. Extending ELFlex beyond x86-64 makes it relevant to a broader range of instrumentation scenarios.

Current Architecture Accommodation:

- The component-based design isolates architecture-specific code
- Abstract interfaces (`generate_trampoline` , `apply_patch`) can have multiple implementations
- The data model is largely architecture-agnostic

Extension Requirements:

1. **Architecture Abstraction Layer** with plugin interface:

```c
typedef struct {
    const char* name;                    // "x86_64", "aarch64", "riscv64"
    uintptr_t (*detect_arch)(elf_binary_t*); // Auto-detect from ELF
    trampoline_t* (*gen_trampoline)(uint64_t, uint64_t);
    unsigned char* (*relocate_prologue)(unsigned char*, size_t);
    int (*fixup_pic)(unsigned char*, uint64_t, uint64_t);
    unsigned char* (*generate_nop_sled)(size_t);
    size_t max_instruction_size;         // For patch planning
} arch_plugin_t;
```

2. **ARM/AArch64-Specific Components**:
   - **Instruction Set Analysis**: ARM's fixed 4-byte instructions simplify patching
   - **Branch Range Handling**: ARM branches use different offset encodings
   - **Register Convention**: Different register roles (x0-x7 for args, x0 for return)
   - **PIC Handling**: ARM uses literal pools rather than RIP-relative addressing
3. **Architecture Detection and Routing**:

```
Algorithm: Multi-Architecture Instrumentation
1. elf_load() detects e_machine field in ELF header
2. Load appropriate architecture plugin
3. Route all arch-specific operations through plugin
4. Generate architecture-appropriate trampolines
5. Apply architecture-aware patching
```

Challenges and Considerations:

- **Instruction Alignment**: ARM requires 4-byte alignment, x86 does not
- **Branch Range Differences**: ARM B instruction has ±128MB range vs x86's ±2GB
- **Conditional Execution**: ARM has conditional instructions, requiring careful analysis
- **Thumb Mode**: ARM's Thumb (16-bit) vs ARM (32-bit) mode switching

> **ADR: Compile-Time vs. Runtime Architecture Selection**
>
> - **Context**: ELFlex must decide how to handle multiple CPU architectures
> - **Options Considered**:
>   1. **Compile-Time Selection**: Separate binaries for each architecture
>   2. **Runtime Plugin System**: Single binary with dynamic architecture modules
>   3. **Fat Binary**: Single executable containing code for all supported arches
> - **Decision**: Runtime plugin system with compile-time architecture detection
> - **Rationale**: Plugin system allows adding new architectures without recompiling core, while compile-time detection ensures only needed code is included in distribution
> - **Consequences**: More complex build system, but flexible deployment and easier testing of new architectures

Architecture Support Strategy	Pros	Cons	Recommended For
Compile-Time Selection	Simple, no runtime overhead	Multiple binaries to distribute	Embedded targets
Runtime Plugin System	Flexible, extensible	Plugin loading complexity	Desktop/server tools
Fat Binary	Single distribution file	Large binary size, platform-specific	macOS-style tools

Higher-Level API and Domain-Specific Languages

Mental Model: The Instrumentation Blueprint Designer

> Currently using ELFlex is like writing machine code—precise but laborious. A higher-level API would be like using CAD software—you describe *what* you want to instrument, and the system figures out *how* to do it safely and correctly.

While ELFlex provides powerful low-level primitives, most instrumentation tasks follow common patterns. A higher-level abstraction can make the tool accessible to developers without deep binary expertise.

Current Architecture Accommodation:

- The existing components are well-factored and can be composed
- Clear interfaces allow building abstraction layers on top
- The monolithic tool design makes integration straightforward

Extension Requirements:

1. **Instrumentation Policy DSL** for declarative hook specification:

```yaml
# Example policy file
instrumentations:
  - function: "malloc"
    hook_type: "entry"
    callback: "malloc_logger"
    parameters:
      - arg1: "size_t size"
    actions:
      - log: "malloc(%zu) called"
      - condition: "size > 1024*1024"
        log: "Large allocation detected"

  - function: "free"
    hook_type: "exit"
    callback: "free_validator"
    actions:
      - validate: "ptr != NULL"
```

2. **Policy Compiler** that transforms DSL to concrete patches:

```
Policy Compilation Pipeline:
1. Parse policy file → Abstract Syntax Tree
2. Resolve symbols in target binary
3. Plan patches for each instrumentation
4. Generate trampoline code with parameter extraction
5. Compile callback functions to PIC
6. Inject callbacks and apply patches
```

3. **Callback Helper Library** for common instrumentation tasks:

```c
// Example helper functions

void log_function_entry(const char* func_name, uint64_t* args, int argc);

void trace_argument_value(int arg_index, uint64_t value);

void modify_return_value(uint64_t* return_slot, uint64_t new_value);

void measure_execution_time(clock_t* start_storage);
```

4. **Interactive REPL** for exploratory instrumentation:

```
ELFlex> attach --pid 1234
ELFlex> list-functions | grep "libc.*alloc"
ELFlex> hook --function malloc --entry --callback log_malloc
ELFlex> watch --function free --exit --print-return-value
ELFlex> detach --keep-hooks
```

Challenges and Considerations:

- **Type Information Loss**: Binary lacks source-level type info for parameters
- **Callback Safety**: Injected code must be carefully isolated
- **Performance Monitoring**: Continuous logging can significantly impact performance
- **Policy Validation**: Must ensure policies don't create circular hooks or deadlocks

Advanced Code Injection Strategies

Mental Model: The Urban Planner for Binary Space

> Current code injection is like building on vacant land. Advanced strategies include "renovating" unused spaces (code caves), "building upward" with overlapping segments, and "prefabricating" injection payloads for rapid deployment.

While Milestone 3 implements basic section injection, several advanced techniques can make instrumentation more stealthy, efficient, or compatible.

Extension Areas:

1. **Code Cave Utilization** - Finding and using existing padding in binaries:

```
Code Cave Detection Algorithm:
1. Scan all LOAD segments for executable regions
2. Identify sequences of 0x00, 0x90 (NOP), or 0xCC (INT3)
3. Calculate cave size and alignment constraints
4. Validate cave is not referenced by any code (no jumps to it)
5. Inject trampolines into caves to avoid section table changes
```

2. **Segment Overlapping and Padding Reuse**:

```c
typedef struct {

    uint64_t file_offset;     // Location in binary file

    uint64_t size;            // Available bytes

    uint64_t virtual_addr;    // Runtime address

    int is_executable;        // Whether region has X permission

    const char* source;       // ".text.padding", ".rodata.align", etc.

} padding_region_t;
```

3. **Position-Independent Code (PIC) Optimization**:

- **Relative Addressing Only**: Ensure injected code uses only RIP-relative addressing
- **Global Offset Table (GOT) Integration**: Share target binary's GOT for efficiency
- **Lazy Binding Stubs**: Generate PLT-like stubs for external function calls

4. **Stealth Injection Techniques**:
   - **Section Header Stripping**: Remove injection evidence after patching
   - **Checksum Preservation**: Maintain original binary checksums where possible
   - **Anti-Disassembly Tricks**: Use opaque predicates and code obfuscation

Challenges and Considerations:

- **Relocation Complexity**: Code caves may be too small for full relocation tables
- **Address Space Layout Randomization (ASLR)**: Injected code must work at any load address
- **Anti-Virus Detection**: Certain injection patterns trigger security software
- **Debugger Compatibility**: Some techniques confuse debuggers and disassemblers

Runtime Instrumentation Enhancements

Mental Model: The Live System Surgeon with Advanced Tools

> Current runtime patching is like emergency surgery—effective but traumatic. Enhanced runtime instrumentation would be like minimally invasive robotic surgery—precise, controlled, and with real-time monitoring.

Milestone 4 establishes basic ptrace-based runtime patching, but several enhancements can improve reliability, performance, and capabilities.

Extension Areas:

1. **Multi-Threaded Process Support**:

```
Safe Multi-Threaded Patching Protocol:
1. Freeze all threads using PTRACE_INTERRUPT
2. Wait for each thread to report stopped state
3. Verify all threads are in safe locations (not in to-be-patched code)
4. Apply patches to shared code pages
5. Flush instruction cache for all threads
6. Resume all threads simultaneously
```

2. **Hot-Swappable Hooks** - Runtime hook management:

```c
// Runtime hook management API

int install_runtime_hook(pid_t pid, const char* symbol,

                         hook_callback_t callback);

int uninstall_runtime_hook(pid_t pid, hook_id_t id);

int list_active_hooks(pid_t pid, hook_info_t** hooks);

int pause_hooks(pid_t pid);    // Temporarily disable

int resume_hooks(pid_t pid);   // Re-enable
```

3. **Performance-Conscious Instrumentation**:
   - **Sampling Mode**: Hook only every Nth invocation to reduce overhead
   - **Conditional Activation**: Enable hooks only when specific conditions met
   - **Lazy Trampoline Generation**: Generate trampolines on first hit, not attach

4. **Kernel-Assisted Patching** (via `perf_event_open` or eBPF):
   - Use kernel facilities for lower-overhead instrumentation

- Leverage hardware performance counters for sampling
- Integrate with existing tracing infrastructure (ftrace, BPF)

Challenges and Considerations:

- **Thread Synchronization**: Ensuring no thread executes code during modification
- **Signal Handling**: Managing pending signals during patching window
- **Resource Limits**: ptrace may be restricted by YAMA or other security modules
- **Performance Overhead**: Continuous instrumentation can significantly slow targets

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option	Reasoning
Inline Hooking	Basic instruction replacement with register save/restore	Full context analysis with stack frame reconstruction	Start simple, add analysis as needed
Multi-Architecture	Separate code paths with `#ifdef ARCH_X86_64`	Runtime plugin system with dlopen()	Plugins allow adding arches without recompile
High-Level API	Simple configuration file (JSON/INI)	Full DSL with parser generator (flex/bison)	JSON is easier initially; DSL for complex policies
Code Cave Detection	Scan for NOP sleds only	Full control flow analysis to find unused code	NOP scanning is 80% effective with 20% effort
Runtime Management	Simple hook tracking in array	Hook database with persistence and versioning	Array works for small numbers; DB for production

Recommended File/Module Structure for Extensions

```
elflex-project/
├── src/
│   ├── core/                   # Existing core components
│   │   ├── elf_loader.c
│   │   ├── patch_planner.c
│   │   ├── trampoline_generator.c
│   │   ├── code_injector.c
│   │   └── runtime_patcher.c
│   ├── extensions/             # New extension modules
│   │   ├── inline_hooking.c    # Inline hook implementation
│   │   ├── exit_hooking.c      # Function exit instrumentation
│   │   ├── arch/               # Architecture plugins
│   │   │   ├── x86_64.c        # Existing x86-64 implementation
│   │   │   ├── aarch64.c       # ARM 64-bit support
│   │   │   ├── arm.c           # ARM 32-bit support
│   │   │   └── plugin.c        # Plugin loading infrastructure
│   │   ├── dsl/                # High-level API components
│   │   │   ├── policy_parser.c  # DSL/JSON parser
│   │   │   ├── policy_compiler.c # Transforms policy to patches
│   │   │   └── callback_lib.c   # Helper library for hooks
│   │   └── advanced_injection.c # Code cave detection, etc.
│   ├── include/                # Headers
│   └── main.c                  # Command-line interface
├── tools/                      # Support tools
│   ├── elflex-repl/            # Interactive REPL
│   └── policy-compiler/        # Standalone policy compiler
└── test/                       # Tests for extensions
    ├── test_inline_hooking.c
    ├── test_multiarch.c
    └── test_policy_dsl.c
```

Infrastructure Starter Code for Architecture Plugins

```c
/* src/extensions/arch/plugin.h - Architecture plugin interface */

#ifndef ARCH_PLUGIN_H

#define ARCH_PLUGIN_H


#include "../../include/elflex.h"


typedef struct arch_plugin arch_plugin_t;


struct arch_plugin {

    const char* name;

    uint16_t elf_machine_id;  // EM_X86_64, EM_AARCH64, etc.



    /* Trampoline generation */

    trampoline_t* (*generate_trampoline)(uint64_t target_addr,

                                         uint64_t hook_addr);



    /* Instruction analysis and patching */

    size_t (*get_instruction_length)(const uint8_t* code);

    int (*is_valid_instruction)(const uint8_t* code, size_t size);

    unsigned char* (*generate_nop_sled)(size_t count);



    /* PIC handling */

    int (*contains_pic_reference)(const uint8_t* instr, size_t size);

    uint8_t* (*fixup_pic_instruction)(uint8_t* instr, size_t size,

                                      uint64_t old_ip, uint64_t new_ip);



    /* Calling convention info */

    const char** argument_registers;  // NULL-terminated array

    const char* return_register;

    size_t stack_argument_offset;     // Where stack args begin



    /* Architecture-specific constants */

    size_t instruction_alignment;

    size_t max_instruction_size;

    uint8_t nop_instruction[8];       // Canonical NOP sequence

    size_t nop_instruction_size;
```

```
};

/* Plugin registry */
int arch_plugin_register(const arch_plugin_t* plugin);
const arch_plugin_t* arch_plugin_for_elf(const elf_binary_t* binary);
const arch_plugin_t* arch_plugin_for_name(const char* name);

/* Built-in plugins */
extern const arch_plugin_t x86_64_plugin;
extern const arch_plugin_t aarch64_plugin;

#endif /* ARCH_PLUGIN_H */
```

```c
/* src/extensions/arch/plugin.c - Plugin registry implementation */

#include "plugin.h"

#include <string.h>


#define MAX_PLUGINS 8

static const arch_plugin_t* registered_plugins[MAX_PLUGINS];

static size_t plugin_count = 0;


int arch_plugin_register(const arch_plugin_t* plugin) {

    if (plugin_count >= MAX_PLUGINS) {

        return ERR_OUT_OF_MEMORY;

    }



    registered_plugins[plugin_count++] = plugin;

    return ERR_SUCCESS;

}


const arch_plugin_t* arch_plugin_for_elf(const elf_binary_t* binary) {

    if (!binary || !binary->ehdr) {

        return NULL;

    }



    uint16_t machine = binary->ehdr->e_machine;



    for (size_t i = 0; i < plugin_count; i++) {

        if (registered_plugins[i]->elf_machine_id == machine) {

            return registered_plugins[i];

        }

    }



    return NULL;

}


const arch_plugin_t* arch_plugin_for_name(const char* name) {

    if (!name) return NULL;



    for (size_t i = 0; i < plugin_count; i++) {
```

```
        if (strcmp(registered_plugins[i]->name, name) == 0) {
            return registered_plugins[i];
        }
    }

    return NULL;
}

/* Plugin initialization - called at program start */
void arch_plugins_init(void) {
    arch_plugin_register(&x86_64_plugin);
    /* Other plugins registered conditionally via #ifdef */
}
```

Core Logic Skeleton for Policy DSL Compiler

```c
/* src/extensions/dsl/policy_compiler.c - Policy to patch translation */

#include "policy_compiler.h"


typedef struct {

    const char* function_name;

    hook_type_t type;           // ENTRY, EXIT, INLINE

    const char* callback_name;

    argument_spec_t* arguments; // Array of argument specifications

    condition_t* conditions;    // Conditional hook application

    action_t* actions;          // Log, modify, etc.

} instrumentation_spec_t;


/* Main compilation function */

error_t* compile_policy_to_patches(const char* policy_file,

                                   elf_binary_t* target,

                                   patch_site_t*** patches,

                                   size_t* patch_count) {

    // TODO 1: Parse policy file into instrumentation_spec_t array

    // TODO 2: For each instrumentation spec:

    //   a. Resolve function_name to address using elf_get_symbol_address

    //   b. Resolve callback_name (may need to compile from source or load from library)

    //   c. Determine patch type based on hook_type_t

    //   d. Generate appropriate trampoline for hook type

    //   e. Plan patch at function address (or specified offset for inline)

    //   f. Compile callback code to PIC if needed

    //   g. Plan code injection for callback if not already present

    // TODO 3: Collect all patches into array

    // TODO 4: Return patches and count for application phase

    // TODO 5: Handle errors gracefully with specific error messages



    error_t* err = error_create();

    // Implementation goes here

    return err;

}


/* Helper: Compile C callback to position-independent machine code */
```

```
unsigned char* compile_callback_to_pic(const char* c_source,
                                       const char* function_name,
                                       size_t* code_size,
                                       error_t* err) {
    // TODO 1: Write C source to temporary file
    // TODO 2: Invoke compiler (gcc/clang) with PIC flags
    // TODO 3: Extract .text section from compiled object file
    // TODO 4: Apply relocations to make code truly position-independent
    // TODO 5: Clean up temporary files
    // TODO 6: Return PIC code and size


    return NULL;
}
```

Language-Specific Hints for C Extensions

1. **Plugin Architecture**: Use `__attribute__((constructor))` for automatic plugin registration:

```c
__attribute__((constructor))
static void init_plugin(void) {
    arch_plugin_register(&my_arch_plugin);
}
```

2. **DSL Parsing**: Consider using libyaml for YAML or jansson for JSON parsing:

```c
// With jansson for JSON policies
json_t* root = json_load_file(policy_file, 0, &error);
json_t* instrs = json_object_get(root, "instrumentations");
size_t index;
json_t* value;
json_array_foreach(instrs, index, value) {
    const char* func = json_string_value(json_object_get(value, "function"));
    // Process each instrumentation
}
```

3. **Multi-Threaded Runtime Patching**: Use `PTRACE_INTERRUPT` and `/proc/[pid]/task/`:

```c
DIR* task_dir = opendir("/proc/1234/task");
struct dirent* entry;
while ((entry = readdir(task_dir)) != NULL) {
    if (entry->d_name[0] != '.') {
        pid_t tid = atoi(entry->d_name);
        ptrace(PTRACE_INTERRUPT, tid, NULL, NULL);
    }
}
```

4. **Code Cave Detection**: Use interval trees for efficient padding region management:

```c
typedef struct interval_node {
    uint64_t start, end;     // [start, end) address range
    uint64_t size() { return end - start; }
    struct interval_node *left, *right;
} interval_node_t;

interval_node_t* find_largest_cave(interval_node_t* root,
                                   uint64_t min_size);
```

Milestone Checkpoint for Extensions

Verification Command for Architecture Plugins:

```bash
# Test x86_64 plugin
$ ./elflex --arch x86_64 --hook malloc --target test_binary_x86_64
[INFO] Using x86_64 architecture plugin
[INFO] Successfully hooked malloc at 0x401050

# Test ARM plugin (if compiled)
$ ./elflex --arch arm --hook malloc --target test_binary_arm
[INFO] Using ARM architecture plugin
[INFO] Successfully hooked malloc at 0x10000

# Verify with disassembly
$ objdump -d patched_binary | grep -A5 "<malloc>"
```

Expected Output for Policy DSL:

```
$ cat policy.yaml
instrumentations:
  - function: "printf"
    hook_type: "entry"
    callback: "log_printf"
    actions:
      - log: "printf called with format: %s"

$ ./elflex --policy policy.yaml --target test_binary
[INFO] Parsing policy from policy.yaml
[INFO] Resolved printf to address 0x401230
[INFO] Compiled callback log_printf (342 bytes PIC)
[INFO] Injected callback at 0x403000
[INFO] Applied hook at printf
[INFO] 1 instrumentation applied successfully

$ ./patched_binary
[HOOK] printf called with format: "Hello, %s!\n"
Hello, world!
```

Debugging Tips for Extensions:

Symptom	Likely Cause	How to Diagnose	Fix
Inline hook crashes after few instructions	Context not fully saved	Single-step through trampoline, check register values before/after	Save all registers and flags used by original instructions
ARM hook works in QEMU but not real hardware	Instruction cache not flushed	Check if `__builtin___clear_cache` is called	Add cache flush after code modification on ARM
Policy file parsed but no hooks applied	Symbol resolution failed	Add debug logging to `elf_get_symbol_address`	Check symbol table presence, use `nm target_binary`
Multi-threaded patching hangs one thread	Thread in uninterruptible state	Check `/proc/pid/task/*/status` for thread state	Skip threads in kernel mode or add timeout
Injected code works once then crashes	PIC assumptions incorrect	Disassemble injected code, check for absolute addresses	Use `-fPIC -fPIE` when compiling callbacks

Glossary

> **Milestone(s):** 1 (Instruction-level Patching), 2 (Function Hooking with Trampolines), 3 (Code Injection), 4 (Runtime Patching via ptrace)

The **Glossary** provides definitive explanations for key terms, acronyms, and concepts used throughout this design document. Binary instrumentation combines concepts from operating systems, compilers, and executable file formats, creating specialized terminology that may be unfamiliar. This reference serves as a centralized dictionary to ensure consistent understanding across all design sections.

Terminology Reference

The following table lists critical terms alphabetically with precise definitions and notes on their relevance to ELFlex.

Term	Definition	Relevance to ELFlex
Address Space Layout Randomization (ASLR)	A security feature that randomizes the virtual memory addresses where system components and user processes are loaded, making memory-based attacks more difficult.	ELFlex must handle ASLR when performing runtime patching, as symbol addresses in a running process will differ from those in the static ELF file. The tool may need to read `/proc/pid/maps` to determine actual load addresses.
Binary Instrumentation	The technique of modifying compiled executable programs (binaries) to insert additional monitoring code or alter behavior, without access to original source code.	This is the core purpose of ELFlex. The tool implements both static binary rewriting (modifying files on disk) and dynamic binary instrumentation (modifying running processes).
Code Cave	Unused space (typically filled with zeros or padding) within existing binary segments that can be repurposed to host injected code without expanding the binary size.	ELFlex's `allocate_code_cave` function searches for such regions as potential injection targets in Milestone 3, though the primary approach is to add new sections.
Dynamic Binary Instrumentation (DBI)	Runtime modification of executing code, typically by intercepting execution at specific points and injecting instrumentation code while the process runs.	ELFlex's runtime patching component (Milestone 4) implements DBI using `ptrace`. This contrasts with static rewriting, which modifies the binary file before execution.
ELF (Executable and Linkable Format)	The standard binary file format for executables, object code, shared libraries, and core dumps on Unix-like systems. Defines headers, sections, segments, and metadata.	ELFlex exclusively targets ELF binaries. The `elf_binary_t` structure and related parsing functions form the foundation for all modification operations.
ELF Structural Integrity	The property that an ELF file's headers, sections, and segments remain valid and consistent after modification, conforming to the ELF specification.	A core challenge throughout ELFlex. Every modification must maintain this integrity—headers must be updated, offsets recalculated, and alignment preserved to produce a loadable binary.
Endianness	The byte order used to store multibyte numerical values in memory (big-endian: most significant byte first; little-endian: least significant byte first).	ELFlex must handle the target's endianness when reading/writing ELF headers and instruction encodings. x86-64 uses little-endian, but the tool should be aware of this for cross-architecture future extensions.
Epilogue Instrumentation	A variant of function hooking that intercepts function exit paths (the epilogue) rather than entry points.	Mentioned as a future extension. ELFocus currently focuses on entry hooking, but the trampoline architecture could be extended to support exit instrumentation.
Function Hooking	Intercepting function calls to execute custom code (a hook) before, after, or instead of the original function.	The central capability of Milestone 2. ELFlex replaces function prologues with jumps to trampolines that call hook functions while preserving original behavior.
Inline Hooking	Inserting hooks within the function body rather than at its entry point, allowing interception at arbitrary locations.	A future extension mentioned. Inline hooking is more complex than entry hooking because it must handle partial instruction replacement and control flow preservation.
Instruction Boundary	The starting memory address of a complete machine instruction. Patches must begin and end at instruction boundaries to avoid creating invalid partial instructions.	Critical for `plan_patch`. ELFlex must disassemble or use instruction length tables to ensure patches don't split multi-byte instructions.
Instruction Cache	A CPU cache that stores recently executed instructions to speed up fetch/decode. After modifying code in memory, the cache may hold stale instructions.	In runtime patching (Milestone 4), ELFlex must flush the instruction cache using `__builtin___clear_cache` or equivalent to ensure the CPU executes the new code.
Memory Protection	Operating system mechanism controlling access permissions (read, write, execute) for memory pages via `mprotect` and page tables.	Runtime patching requires changing code page protections from read-execute to read-write-execute before modification, then restoring them. Handled by `make_page_writable`.
Module Isolation	A design principle where components are kept independent with clear interfaces, minimizing coupling and allowing separate development and testing.	ELFlex's architecture follows this principle: the ELF Loader, Patch Planner, Trampoline Generator, Code Injector, and Runtime Patcher are separate components with defined APIs.

Term	Definition	Relevance to ELFlex
Monolithic Tool	A single integrated executable rather than a plugin-based framework. All functionality is compiled into one binary.	ELFlex is designed as a monolithic tool for simplicity, though the architecture could evolve to support plugins (e.g., `arch_plugin_t` for multiple architectures).
NOP Sled	A sequence of no-operation instructions (NOPs) used as padding to fill space when replacing instructions with smaller ones, maintaining code size and alignment.	Generated by `generate_nop_sled` in Milestone 1. When patching an instruction with a smaller one, ELFlex fills the remainder with NOPs to preserve subsequent instruction offsets.
Ownership Hierarchy	A clear specification of which data structure owns each allocated memory block, dictating responsibility for allocation and deallocation to prevent memory leaks.	Critical in C implementation. Each `elf_binary_t`, `patch_site_t`, `trampoline_t`, etc., has defined ownership: the creator must free it, and child pointers are typically owned by the parent structure.
Position-Independent Code (PIC)	Code that can execute correctly regardless of its absolute load address, using relative addressing and Global Offset Table (GOT) indirection instead of absolute addresses.	Injected code in Milestone 3 must be PIC (or relocated) because its load address isn't known until injection time. ELFlex's `compile_callback_to_pic` produces such code.
Prologue	The initial sequence of instructions at a function's entry that sets up the stack frame, saves callee-saved registers, and allocates local variables.	ELFlex's trampoline relocation targets the prologue. The tool saves the original prologue (first 5-20 bytes), replaces it with a jump, and executes the relocated prologue in the trampoline.
Ptrace	A Linux system call (`ptrace`) that allows one process (the tracer) to observe and control another process (the tracee), including reading/writing memory and registers.	The foundation of ELFlex's runtime patching (Milestone 4). Functions like `attach_to_process` , `read_process_memory` , and `write_process_memory` wrap `ptrace` operations.
Relative Jump	A branch instruction where the target address is encoded as a signed offset from the current instruction pointer (RIP on x86-64), rather than an absolute address.	When ELFlex patches a function prologue with a jump to a trampoline, it typically uses a relative jump (`E9` on x86-64). This has limited range (±2GB), sometimes requiring indirect jumps.
RIP-Relative Addressing	An x86-64 addressing mode where memory operands are specified as an offset from the instruction pointer (RIP), commonly used for accessing global data and function pointers.	When ELFlex relocates a prologue containing RIP-relative instructions (like `mov rax, [rip+0x1234]`), it must adjust the offset via `fixup_rip_relative` because the instruction's new RIP differs.
Runtime Instrumentation	Modifying executing code in a running process, as opposed to static binary rewriting. Combines debugging interfaces with code injection.	Implemented in Milestone 4 using `ptrace` . ELFlex attaches to a process, pauses it, modifies its memory to install trampolines, then resumes it with hooks active.
Segment	A memory region described by a program header (`Elf64_Phdr`) that the operating system loader maps into the process address space. Segments contain one or more sections.	ELFlex must handle segment boundaries when injecting code. Adding a new executable section may require creating a new `PT_LOAD` segment or expanding an existing one.
Segment Overlap	When two ELF segments claim overlapping virtual address ranges, which is invalid and causes loader errors.	A pitfall during code injection (Milestone 3). When shifting segments to make space, ELFlex must ensure no overlap occurs by updating all `p_vaddr` and `p_paddr` fields.
Section	A contiguous chunk of the ELF file with a specific purpose (code, data, symbols, strings) described by a section header (`Elf64_Shdr`). Sections are logical divisions used by linkers.	ELFlex manipulates sections extensively: reading `.text` for patching, `.symtab` for symbols, and creating new sections (`.inject`) for injected code.
Static Binary Rewriting	Offline modification of binary files on disk before execution, as opposed to runtime instrumentation.	ELFlex implements static rewriting in Milestones 1-3: loading an ELF file, applying patches/trampolines, injecting sections, and writing a modified binary to disk.

Term	Definition	Relevance to ELFlex
Symbol Table	An ELF section (`.symtab` or `.dynsym`) containing mappings between function/variable names and their addresses (symbols), used by linkers and debuggers.	ELFlex uses the symbol table via `elf_get_symbol_address` to locate function entry points for hooking. The tool may also need to add symbols for injected functions.
Syscall Injection	A technique where a debugger forces a traced process to execute a system call (like `mprotect`) by manipulating its registers and instruction pointer.	Used in runtime patching to change memory protection. ELFlex's `make_page_writable` injects an `mprotect` syscall into the target process via `ptrace` to make code pages writable.
Trampoline	A small code sequence that redirects execution flow. In instrumentation, a trampoline typically saves processor state, calls a hook function, restores state, then jumps to the original code.	Core to Milestone 2. ELFlex's `trampoline_t` contains generated trampoline code that handles the detour: save registers → call hook → restore registers → execute relocated prologue → jump to original function.
YAMA ptrace_scope	A Linux security module (`kernel.yama.ptrace_scope`) that restricts `ptrace` usage to prevent arbitrary process debugging, typically allowing only parent-child debugging.	A runtime patching pitfall. ELFlex may fail with `ERR_PTRACE_PERM` if YAMA restricts attachment. The user may need `sudo` or to adjust `/proc/sys/kernel/yama/ptrace_scope` .

Additional Project-Specific Terminology

Term	Definition	Relevance to ELFlex
Architecture Plugin	A module (modeled by `arch_plugin_t`) providing architecture-specific implementations for instruction encoding, trampoline generation, and other CPU-dependent operations.	A future extension concept. ELFlex initially targets x86-64 but could support ARM via plugins. The plugin would provide machine-specific functions like `generate_trampoline`.
Assembly Line	The analogy used in High-Level Architecture: ELFlex's components act like stations in an assembly line, each transforming the binary representation in sequence.	Helps conceptualize the data flow: ELF Loader (unpack) → Patch Planner (mark) → Trampoline Generator (build) → Code Injector (extend) → Binary Writer (repack).
Dry-Run	An operational mode where ELFlex shows what changes would be made without actually modifying files or processes, useful for validation and debugging.	Suggested as a future feature. Would involve running the full patching pipeline but skipping final write/`ptrace_pokedata` steps, instead logging planned changes.
Fail Early, Fail Cleanly	The design philosophy that ELFlex should detect errors as soon as possible and ensure it never leaves the target binary or process in an unrecoverable state.	Guides error handling throughout. For runtime patching, this means saving original code before modification so changes can be reverted if later steps fail.
Fixture	Pre-prepared test data or binary used in unit and integration tests to verify ELFlex's behavior on known inputs.	Part of the testing strategy. ELFlex's test suite uses simple fixture binaries (e.g., `test_fixture_simple`) to validate patching without complex dependencies.
Hot-Swappable Hooks	Hooks that can be installed and uninstalled at runtime without restarting the instrumented process, allowing dynamic instrumentation control.	A future extension concept. Would require maintaining original code copies and toggling between patched and unpatched states via runtime patching.
Interval Tree	A data structure (`interval_node_t`) for efficiently storing and querying non-overlapping address ranges, used to track modified regions in a binary.	Could be used internally by Patch Planner to prevent patch overlap (`ERR_PATCH_OVERLAP`) by tracking which address ranges have already been modified.
Latent Errors	Errors that don't cause immediate failure in ELFlex but manifest later when the modified binary executes, such as incorrect RIP-relative fixups or misaligned jumps.	A significant concern. ELFlex uses validation functions (`validate_patch`, `validate_trampoline`) and testing to catch latent errors before they cause runtime crashes.
Memory Ownership	The clear specification (following the ownership hierarchy) of which structure owns each allocated memory block, dictating responsibility for deallocation.	Critical for preventing memory leaks. For example, `elf_binary_t` owns the `buffer` and all `section_t` / `segment_t` structures; patches own their `new_bytes` arrays.
Multi-Threaded Process Complications	Challenges when patching a process with multiple threads: pausing one thread doesn't pause others, which may be executing code being modified.	A runtime patching pitfall. ELFlex currently assumes single-threaded targets or patches when all threads are in safe states (e.g., at synchronization points).
Policy DSL	A Domain-Specific Language for declaring instrumentation policies (which functions to hook, what callbacks to run, conditions, and actions).	A future extension concept. The `compile_policy_to_patches` function would translate high-level policies into concrete `patch_site_t` specifications.
Plugin Registry	A system for managing architecture-specific implementations (`arch_plugin_t`), allowing runtime selection of the appropriate plugin based on `e_machine`.	A future extension concept. Would maintain a table of plugins keyed by `ELF_MACHINE` constants (`EM_X86_64`, `EM_AARCH64`) for multi-architecture support.
Sampling Mode	Instrumentation that activates only periodically (e.g., every 1000th call) to reduce overhead, as opposed to hooking every invocation.	A future optimization. Could be implemented by having the trampoline maintain a counter and only calling the hook when the counter reaches a threshold.

Term	Definition	Relevance to ELFlex
Segment Overlapping	Using padding within existing segments (code caves) for code injection rather than adding new segments, minimizing binary size increase.	An alternative approach considered in Milestone 3. ELFlex primarily adds new sections for clarity, but `allocate_code_cave` supports overlapping when space exists.
Stealth Injection	Injection techniques designed to avoid detection by security software or integrity checks, such as avoiding new segments or hiding modifications.	Not a goal for ELFlex (which is a learning tool), but mentioned as a concept in future extensions for real-world instrumentation scenarios.
Surgical Rollback Protocol	The metaphor for runtime patching cleanup: saving original state before modification so changes can be reverted if errors occur, leaving the process intact.	Embodies the "fail cleanly" philosophy. ELFlex saves original bytes before writing trampolines and could restore them if injection fails before completion.
Test Runner	Infrastructure for executing and verifying tests (`test_suite_t` , `run_test_case`), automating validation of ELFlex's functionality across milestones.	Implemented in the testing strategy. The test runner executes test binaries, compares outputs, and reports success/failure for each milestone's acceptance criteria.
Validation Mode	An operational mode that performs extensive checks (ELF integrity, patch validity, symbol resolution) without making modifications.	Suggested as a future feature. Would run all planning and validation steps but skip the actual write/ `ptrace_pokedata` , reporting any issues found.
Workshop Layout	The analogy for ELFlex's directory structure: different "workbenches" (directories) for different tasks (parsing, patching, injection, runtime).	Guides the recommended file/module structure: `src/elf/` for parsing, `src/patch/` for patching, `src/trampoline/` for trampolines, etc., separating concerns clearly.

Implementation Guidance

Since the Glossary is a reference section without executable components, no implementation code is provided. However, maintaining consistency in terminology is critical for code readability and documentation. Consider these practices:

1. **Use Consistent Naming**: Always use the exact terms from this glossary when naming variables, functions, and comments. For example, use "trampoline" not "stub" or "detour," and "prologue" not "function header."
2. **Glossary Integration**: Include a brief version of this glossary in your project's README or documentation to help contributors understand the domain language.
3. **Code Comments**: When implementing functions that manipulate these concepts, add a brief comment referencing the glossary term:

```c
/* Generate a trampoline (see Glossary) for function hooking. */
trampoline_t* generate_trampoline(uint64_t target_addr, uint64_t hook_addr) {
    // Implementation...
}
```

4. **Error Messages**: Use glossary terms in error messages to be precise:

```c
error_set(err, ERR_INVALID_OFFSET,
          "Patch offset not within executable .text section",
          "Ensure target is at an instruction boundary within a loaded segment");
```

5. **Testing Terminology**: When writing tests, name test cases after the concepts they validate:
   - `test_trampoline_register_preservation`
   - `test_code_injection_pic_relocation`
   - `test_runtime_patching_ptrace_attach`

By consistently using this terminology, you create a shared vocabulary that makes the codebase more understandable and maintainable.