Contents

1 arm-fp-emu.c

Called by the dynamic linker/loader and where the instrumentation process begins.

```c
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <link.h>
#include "librunt.h"
#include "maps.h"
#include "dso-meta.h"
#include "relf.h"
#include "debug-print.h"
#include "rmaps.h"
#include "assembly.h"

// Fixes undefined symbols at build stage: the --defsym compiler flag doesn't solve this.
char* __private_strdup(const char *s) { return strdup(s); }
void* __private_malloc(size_t size) { return malloc(size); }

/*
* Sets the write flag for a region of memory.
* A 'guarantee' pointer ensures that this region includes that
* address.
*/
int make_writable(void* from, void* to, void* guarantee) {

    from = ROUND_DOWN_PTR_TO_PAGE(from);
    if (guarantee != NULL && (guarantee < from || guarantee > to)) {
        printfdbg("ERROR: Couldn't make region writable\n");
        return -1;
    }

    size_t len = to - from;
    int perms = PROT_READ | PROT_WRITE | PROT_EXEC;

    printfdbg("mprotect(%p, %d, rwx)\n", from, len);
    int ret = mprotect(from, len, perms);

    if (ret != 0) {
        printfdbg("ERROR: Couldn't make region writable %s\n");
        perror("mprotect");
        return -1;
    }
    return 0;
}

/*
* Returns a pointer to the end of an ELF header
*/
void* end_of_header(void* sections_start) {
    Elf32_Ehdr* header = (Elf32_Ehdr*) sections_start;

    // Elf32_Off is typedef'd in elf.h as uint32_t:
    //  "typedef uint32_t Elf32_Off;"
    uint32_t ph_table_offset = header->e_phoff;
    // Elf32_Half is also uint16_t
    uint16_t ph_table_len = ((uint16_t) header->e_phnum) * ((uint16_t)
        header->e_phentsize);

    return ((int8_t*) sections_start) + ph_table_offset + ph_table_len;
}

/*
* Attempts to set the write flag for a region of memory.
* Returns whether change was successful (0 == success)
*/
int try_set_mem_writable(struct maps_entry *maps_ent, void* seg_start,
    void* seg_end) {
    if (maps_ent->w != 'w') {
        int ret = make_writable(seg_start, seg_end, NULL);
        if (ret == 0) {
            printfdbg("Write perms now set for %p-%p.\n",
                ROUND_DOWN_PTR_TO_PAGE(seg_start), seg_end);
            return 0;
        } else {
            printfdbg("ERROR: Failure to make writable\n");
            exit(1);
        }
    } else {
        printfdbg("Write perms already set (%c%c%c), continuing.\n",
            maps_ent->r, maps_ent->w, maps_ent->x);
        return 1;
    }
}

/*
```

```c
* Core of the instrumentation process.
* Looks through a mapped region in memory and finds a floating-point instruction.
* If one is found, it is replaced by a branch instruction and a trampoline is made and
* written to somewhere in memory.
*/
void replace_instrs_in_segment(struct maps_entry *maps_ent, void* from, void* to) {
    void* sections_start = from;
    void* sections_end = to;
    void* seg_start = maps_ent->first;
    void* seg_end = maps_ent->second;

    printfdbg("\tWe have entered replace_instructions() \n");
    printfdbg("\tRange of executable instructions (inside segment range): \n");
    printfdbg("\t");
    if (seg_start != sections_start) {
       printfdbg("(%p-)",seg_start);
    }
    printfdbg("%p-%p", sections_start, sections_end);
    if (seg_end != sections_end) {
        printfdbg("(-%p)",seg_end);
    }
    printfdbg("\n");

    void* instrs_start = sections_start;
    if (0 == strncmp(sections_start, "\177ELF",4)) {
        // Probably an ELF header so skip it
        instrs_start = end_of_header(sections_start);
        printfdbg("ELF headed spotted at %p, skipping to %p\n", sections_start, instrs_start);
    }

    assert(instrs_start >= seg_start && instrs_start <= seg_end);
    assert(sections_end >= seg_start && sections_end <= seg_end);

    int b_is_writable = maps_ent->w == "w";
    int b_did_perm_change = 0;

    printfdbg("Scanning through %p-%p for FP instructions\n", instrs_start, sections_end);
    for (int8_t* instr = instrs_start; instr < sections_end-4; instr += 2) {
        void* tramp = generate_trampoline(instr);
        if (tramp == NULL) {
            continue;
        } else {
            printfdbg("Trampoline written for FP instruction at %p in %p-%p\n", instr, instrs_start, sections_end);
            printfdbg(" - writing jump at the mentioned instr (%p)\n", instr);
            if (!b_is_writable && 0 == try_set_mem_writable(maps_ent, seg_start, seg_end)) {
                b_is_writable = 1;
                b_did_perm_change = 1;
            }
            insert_probe(instr, tramp);
        }
    }
}

/*
* Callback for librunt.
* Librunt passes information about a mapped region of memory and
* this callback method does some setup then instruments it.
*
* In full transparency, this method resembles code belonging to my supervisor Dr. Stephen Kell.
* His project "libsystrap" uses his other project "librunt" which is also a dependency for this project.
* Not only does some of the code look similar because they are using librunt in mostly the same way because
* they both are rewriting code in memory, but libsystrap contributed to what I used to learn about instrumentation
* and how to use librunt.
* The relevant code is spread throughout the file "src/trap.c" which can be found at the link below. This includes the methods
* "trap_one_executable_region_given_shdrs", "trap_one_executable_region", and "trap_one_instruction_range".
* https://github.com/stephenrkell/libsystrap/blob/21c5b00eb256f5489ee0d163efecc3398dfef2c9/src/trap.c
*/
static int handle_maps_entry(struct maps_entry *maps_ent, char *linebuf, void *arg) {
    // Tests for memory regions that don't need instrumenting.
    // "[" is here to ensure stability by exercising overcaution, but in practice you'd want
    // to be more specific like the rest of the search strings.
    char* to_skip[] = {"[", "[stack]", "[vvar]", "[sigpage]", "[vdso]", "[vectors]", "libm-2.31.so", "libkeystone.so.0", "libcapstone.
```

```
       so.4"};
    for (int i = 0; i < sizeof(to_skip) / sizeof(to_skip[0]); i++) {
        if (NULL != strstr(maps_ent->rest, to_skip[i])) {
            printfdbg("\tSkipping %s\n", maps_ent->rest);
            return 0;
        }
    }
    printfdbg("Handling maps entry for \"%s\"\n", maps_ent->rest);
    printfdbg("\tGetting file metadata.\n");
    struct file_metadata* meta = __runt_files_metadata_by_addr(maps_ent
       ->first);
    if (meta == NULL) {
        printfdbg("File metadata not found for: %s\n", maps_ent->rest);
        return 1;
    } else if (maps_ent->x != 'x') {
        printfdbg("\tNot executable, skipping.\n");
        return 0;
    }

    // "Base address shared object is loaded at" - definition
    ElfW(Addr) l_addr = meta->l->l_addr;

    if (meta->shdrs == NULL) {
        printfdbg("\tNo section headers: file_metadata->shdrs is null.\
           n");
        return 1;
    } else {
        printfdbg("\tFound section headers.\n");
    }

    // Range within segment containing exactly only sections
    void* sections_from = l_addr + find_section_boundary((uintptr_t) (
       maps_ent->first - l_addr), \
            SHF_EXECINSTR, \
            0, \
            meta->shdrs, \
            meta->ehdr->e_shnum, \
            NULL);
    void* sections_to = l_addr + find_section_boundary((uintptr_t) (
       maps_ent->second - l_addr), \
            SHF_EXECINSTR, \
            1, \
            meta->shdrs, \
            meta->ehdr->e_shnum, \
            NULL);

    printfdbg("%s\n", maps_ent->rest);
    assert(maps_ent->first <= sections_from && sections_from <= sections_to && sections_to <= maps_ent->second);
    printfdbg("Maps entry goes from (%p-)%p-%p(-%p)\n", maps_ent->first, sections_from, sections_to, maps_ent->second);

    replace_instrs_in_segment(maps_ent, sections_from, sections_to);
    return 0;
}

/*
* Called by the dynamic linker/loader before the base program.
* This is where the instrumentation happens.
*/
static int entrypoint(void) __attribute__((constructor(101)));
static int entrypoint(void) {
    int fd = open_maps();
    start_disasm_engine();
    start_asm_engine();
    emulator_init();

    // Replace instructions
    process_all_lines(fd, handle_maps_entry);

    close_maps();
    stop_disasm_engine();
    stop_asm_engine();
    return 0;
}
```

2 assembly.h

Where assembly and disassembly takes place as well as methods to write and link up trampolines.

```
#include <keystone/keystone.h>
#include <capstone/capstone.h>
#include <inttypes.h>
#include <sys/mman.h>
#include <stddef.h>
#include "fpuemu.h"
#include <unistd.h>
#include <stdlib.h>
#define PAGE_SIZE sysconf(_SC_PAGE_SIZE)

int8_t MIN_PRINTABLE_ASCII = (int8_t) 0x20; // 32
int8_t MAX_PRINTABLE_ASCII = (int8_t) 0x7F; // 127
int INT24_MIN = -(1 << 23);
int INT24_MAX = (1 << 23) - 1;

/*
* Template for the trampolines used to connect
* a probe site and emulation routine at run-time.
*/
int8_t template[] = {
    0xFF, 0x5F, 0x2D, 0xE9, // push {r0-r12, r14}
    0xAD, 0x5E, 0x0D, 0xE3, // movw r5, #0xdead
    0x05, 0x58, 0xA0, 0xE1, // lsl r5, r5, #16
    0xEF, 0x6E, 0x0B, 0xE3, // movw r6, #0xbeef
    0x06, 0x50, 0x85, 0xE1, // orr r5, r5, r6
    0x00, 0x00, 0xA0, 0xE3, // mov r0, #0
    0x00, 0x10, 0xA0, 0xE3, // mov r1, #0
    0x00, 0x20, 0xA0, 0xE3, // mov r2, #0
    0x00, 0x30, 0xA0, 0xE3, // mov r3, #0
    0x35, 0xFF, 0x2F, 0xE1, // blx r5
    0xFF, 0x5F, 0xBD, 0xE8, // pop {r0-r12, r14}
    0xFE, 0xFF, 0xFF, 0xEA  // b #0
};

/*
* Offsets of various instructions in the trampoline
* template and the names of registers used.
*/
int MOV_UPPER_OFFSET = 1 * 4;
int MOV_LOWER_OFFSET = 3 * 4;
int MOV_R0_OFFSET = 5 * 4;
int MOV_R1_OFFSET = 6 * 4;
int MOV_R2_OFFSET = 7 * 4;
int MOV_R3_OFFSET = 8 * 4;
int RET_OFFSET = 11 * 4;
int REG_SCRATCH = 6;
int REG_CALL = 5;

/*
* Capstone (disassembly framework) and
* Keystone (assembly framework) engine handles.
*/
csh* cs_handle;
ks_engine* ks_handle;

/*
* Copy four bytes from one location into another.
* Used to replace four-byte ARM instructions.
*/
void clobber(void* dst, void* src) {
    printfdbg("clobbering\n");
    make_writable(dst, dst + 4, NULL);
    printfdbg(" - writing src into dst\n");
    memcpy(dst, src, 4);
}

void start_asm_engine() {
    if(ks_open(KS_ARCH_ARM, KS_MODE_ARM, &ks_handle) != KS_ERR_OK) {
        printfdbg("\tUnable start the keystone assembly engine.\n");
    }
}

void stop_asm_engine() {
    ks_close(ks_handle);
}

void start_disasm_engine() {
    if(cs_open(CS_ARCH_ARM, CS_MODE_ARM, &cs_handle) != CS_ERR_OK) {
        printfdbg("\tUnable start the capstone disassembly engine.\n");
        exit(-1);
    }
    cs_malloc(cs_handle);

    // enable full range of disassembly information
    cs_option(cs_handle, CS_OPT_DETAIL, CS_OPT_ON);
```

```c
}

void stop_disasm_engine() {
      cs_close(cs_handle);
}

/*
* Use the Keystone assembler to convert a string in
* assembly to machine-code.
*/
int8_t* assemble_instr(char* assembly) {
    int8_t* instr;
    size_t size;
    size_t count;
    if (ks_asm(ks_handle, assembly, 0, &instr, &size, &count) != KS_ERR_OK) {
        printfdbg("Unable to assemble instruction '%s'\n", assembly);
        exit(-1);
    }
    return instr;
}

/*
* Returns a pointer to a machine-code branch
* instruction (ARM) that branches to the given offset.
* 'offset' is a human-readable string like "0xfae".
*/
int8_t* assemble_branch(char* offset) {
    char assembly[100];
    sprintf(assembly, "b #%s", offset);
    return assemble_instr(assembly);
}

/*
* Returns a pointer to a machine-code 'mov'
* instruction (ARM). E.g. reg = 5 for r5. 'val' is
* the immediate value.
*/
int8_t* assemble_mov(uint8_t reg, uint16_t val) {
    int8_t instr[] = {
        0xEF, 0x1E, 0x0B, 0xE3 // movw r1, #0xbeef
    };

    if (reg > 0xF) {
        printfdbg("assemble_mov() - Register not supported: %lu", reg);
        return NULL;
    }

    // set register
    instr[1] = (reg & 0x0F) << 4;

    // the immediate value is divided and placed in interesting positions
    // (value's nibbles 0,1,2,3 to mov's nibbles 5,3,0,1)
    // so various logical operations are needed
    // Also it is a "byte" array not a nibble array so OR-ing is needed
    instr[2] = (instr[2] & 0xF0) | ((val & 0xF000) >> 12);
    instr[1] = (instr[1] & 0xF0) | ((val & 0x0F00) >> 8);
    instr[0] = val & 0x00FF;

    int8_t* outbuf = malloc(4*sizeof(int8_t));
    memcpy(outbuf, instr, sizeof(instr));
    return outbuf;
}

/*
* Returns a struct that contains information about what instruction
* exists at the provided pointer.
*/
cs_insn* disassemble_instr(void* p_instr) {
    int8_t code[4];
    memcpy(code, (int8_t*) p_instr, 4);

    cs_insn* instr;
    assert(cs_handle != NULL);
    int count = cs_disasm(cs_handle, p_instr, 4, 0, 1, &instr);

    cs_insn* output;
    if (count == 0 || 0 == strcmp(instr[0].mnemonic, "") ) {
        output = NULL;
    } else {
        cs_insn* insn_copy = malloc(sizeof(cs_insn));
        memcpy(insn_copy, &instr[0], sizeof(cs_insn));
        output = insn_copy;
        cs_free(instr, count);
    }

    char* name = output == NULL ? NULL : cs_insn_name(cs_handle, output->id);
    return output;
}

```

```c
/*
* Returns the name of the instruction at the
* provided pointer.
*/
char* instr_name(void* instr){
    cs_insn* disassembly = disassemble_instr(instr);
    if (disassembly == NULL) {
        return NULL;
    }
    char* name = cs_insn_name(cs_handle, disassembly->id);
    char* out = malloc(10);
    strcpy(out, name);
    free(disassembly);
    return out;
}

/*
* In the instrumentation stage, this method displaces the floating-point
* instruction with a branch that points to the start of a pre-written trampoline.
* The trampoline is also written to so that when it returns, it returns to just after
* the displaced FP instruction.
*/
int insert_probe(void* instr, void* tramp) {
    printfdbg("Inserting probe at %p to connect to trampoline at %p\n", instr, tramp);

    // Calculate trampoline offset
    ptrdiff_t offset = ((int8_t*) tramp - (int8_t*) instr);
    ptrdiff_t offset_reverse = (((int8_t*) instr + 4) - ((int8_t*) tramp + RET_OFFSET));
    // Ensure trampoline is close enough for the offset to be written
    assert(INT24_MIN <= offset && offset <= INT24_MAX);
    assert(INT24_MIN <= offset_reverse && offset_reverse <= INT24_MAX);

    // Convert offset into a string to pass to the assembler
    char str_offset[100];
    char str_offset_reverse[100];
    sprintf(str_offset, "%td", offset);
    sprintf(str_offset_reverse, "%td", offset_reverse);
    int8_t* probe_site_to_tramp = assemble_branch(str_offset);
    int8_t* tramp_to_probe_site = assemble_branch(str_offset_reverse);

    #ifdef DO_DBG_PRINT
    char* before = instr_name(instr);
    char* after = instr_name(probe_site_to_tramp);
    char* back_again = instr_name(tramp_to_probe_site);

    printfdbg(" - writing tramp branch into instr at %p\n", instr);
    printfdbg("      - instr: %02x%02x%02x%02x",
        *((int8_t*) instr)&0xff,
        *((int8_t*) instr + 1) & 0xff,
        *((int8_t*) instr + 2) & 0xff,
        *((int8_t*) instr + 3) & 0xff);
    printfdbg(" (%s)\n", before);
    printfdbg("      - assembly: %02x%02x%02x%02x",
        *((int8_t*) probe_site_to_tramp)&0xff,
        *((int8_t*) probe_site_to_tramp + 1) & 0xff,
        *((int8_t*) probe_site_to_tramp + 2) & 0xff,
        *((int8_t*) probe_site_to_tramp + 3) & 0xff);
    printfdbg(" (%s)\n", after);

    printfdbg(" - writing return branch into tramp at %p\n", (int8_t*) tramp + RET_OFFSET);
    printfdbg("      - instr: %02x%02x%02x%02x",
        *((int8_t*) tramp + RET_OFFSET)&0xff,
        *((int8_t*) tramp + RET_OFFSET + 1) & 0xff,
        *((int8_t*) tramp + RET_OFFSET + 2) & 0xff,
        *((int8_t*) tramp + RET_OFFSET + 3) & 0xff);
    printfdbg(" (%s)\n", before);
    printfdbg("      - assembly: %02x%02x%02x%02x",
        *((int8_t*) probe_site_to_tramp) & 0xff,
        *((int8_t*) probe_site_to_tramp + 1) & 0xff,
        *((int8_t*) probe_site_to_tramp + 2) & 0xff,
        *((int8_t*) probe_site_to_tramp + 3) & 0xff);
    printfdbg(" (%s)\n", after);
    #endif

    // Replace FP instruction with branch
    clobber(instr, probe_site_to_tramp);

    // Add return branch to the end of the trampoline
    clobber((int8_t*)tramp + RET_OFFSET, tramp_to_probe_site);

    #ifdef DO_DBG_PRINT
    printfdbg(" - branch written\n");
    printfdbg(" - Therefore, '%s' replaced with '%s' at %p\n", before, after, instr);
    free(before);
    free(after);
    free(back_again);
    #endif

    ks_free(probe_site_to_tramp);
    ks_free(tramp_to_probe_site);
}

/*
* Finds and reserves a page of memory near 'instr_addr'.
* Returns a pointer to the start of this page.
*/
void* mmap_nearby(void* instr_addr) {
    unsigned int perms = PROT_EXEC | PROT_READ | PROT_WRITE;
    unsigned int flags = MAP_FIXED_NOREPLACE | MAP_PRIVATE | MAP_ANONYMOUS;

    void* range_low = 0x0;
    void* range_high = 0xFFFFFFFF;
    if (instr_addr >= -INT24_MIN) {
        range_low = instr_addr + INT24_MIN;
    }
    if (instr_addr <= range_high - INT24_MAX) {
        range_high = instr_addr + INT24_MAX;
    }
    assert(range_low <= range_high);
    void* search_from = ROUND_UP_PTR_TO_PAGE(range_low);
    void* search_to = ROUND_DOWN_PTR_TO_PAGE(range_high);
    printfdbg("mmap_nearby: We have %p-%p range but will look at pages %p-%p\n", range_low, range_high, search_from, search_to);

    assert(search_from <= instr_addr && instr_addr <= search_to);

    void* map_region = NULL;
    printfdbg("Searching from %p to %p\n", search_from, search_to);
    for (int page_start = search_from; page_start < search_to; page_start += PAGE_SIZE) {
        printfdbg("mmap(%p, %d, ...) = ", page_start, sizeof(template));
        map_region = mmap(page_start, sizeof(template), perms, flags, -1, 0);
        printfdbg("%p (should be %p)\n", map_region, page_start);
        if (map_region == page_start) {  // request accepted
            break;
        }
        #ifdef DO_DBG_PRINT
        perror("mmap");
```

```c
        #endif
    }
    if (map_region == MAP_FAILED || map_region == NULL) {
        printfdbg("ERROR: no space for trampoline near instruction %p (see mmap error below)\n", instr_addr);
        #ifdef DO_DBG_PRINT
        perror("mmap");
        #endif
        return NULL;
    }
    return map_region;
}

/*
* Reserves a page of memory and writes the trampoline template to it.
*/
void* gen_template_tramp(void* instr_addr) {
    void* p_template = mmap_nearby(instr_addr);
    if (p_template == NULL) {
        printfdbg("ERROR: no space for trampoline near instruction %p (see mmap error below)\n", instr_addr);
        #ifdef DO_DBG_PRINT
        perror("mmap");
        #endif
        return NULL;
    }
    memcpy(p_template, &template, sizeof(template));
    return p_template;
}

/*
* Shorthand to replace an instruction in a trampoline.
*/
void insert_tramp_instr(void* trampoline, void* instr, int offset) {
    memcpy(((int8_t*)trampoline) + offset, (int8_t*) instr, 4*sizeof(int8_t));
}

/*
* Inserts the emulation routine arguments into a trampoline.
*/
void tramp_insert_emu_args(void* trampoline, int arg1, int arg2, int arg3, int arg4) {
    int8_t* mov_r0 = assemble_mov(0, arg1);
    int8_t* mov_r1 = assemble_mov(1, arg2);
    int8_t* mov_r2 = assemble_mov(2, arg3);
    int8_t* mov_r3 = assemble_mov(3, arg4);
    insert_tramp_instr(trampoline, mov_r0, MOV_R0_OFFSET);
    insert_tramp_instr(trampoline, mov_r1, MOV_R1_OFFSET);
    insert_tramp_instr(trampoline, mov_r2, MOV_R2_OFFSET);
    insert_tramp_instr(trampoline, mov_r3, MOV_R3_OFFSET);
    free(mov_r0);
    free(mov_r1);
    free(mov_r2);
    free(mov_r3);
}

/*
* Writes the address of the emulation routine into the trampoline for
* branching later. As described in the report, the address is divided and
* loaded in two parts into a register by instructions in the trampoline.
*/
void link_tramp_to_emu(void* trampoline, void* func_address) {
    // Split address into two two-byte pieces.
    int16_t word_lower = ((int32_t) func_address) & 0x0000FFFF;
    int16_t word_upper = (((int32_t) func_address) & 0xFFFF0000) >> 16;

    int8_t* mov_instr0 = assemble_mov(REG_CALL, word_upper);
    int8_t* mov_instr1 = assemble_mov(REG_SCRATCH, word_lower);

    if (mov_instr0 == NULL || mov_instr1 == NULL) {
        printfdbg("ERROR: couldn't assemble func address into mov");
        exit(1);
    }

    insert_tramp_instr(trampoline, mov_instr0, MOV_UPPER_OFFSET);
    insert_tramp_instr(trampoline, mov_instr1, MOV_LOWER_OFFSET);

    free(mov_instr0);
    free(mov_instr1);
}

/*
* Returns a pointer to the beginning of the trampoline or NULL if failed.
* In three places there is a hard-coded check for the 'vadd.f32' instruction as
* it is the only instruction emulated so far in the emulator. In a full solution,
* these checks would be removed.
```

```
*/
void* generate_trampoline(void* instr_addr) {
    cs_insn* disassembly = disassemble_instr(instr_addr);

    if (disassembly == NULL) {
        return NULL;
    } else if (disassembly->id != ARM_INS_VADD) {
        free(disassembly);
        return NULL;
    }

    // Get which S registers are used
    cs_arm* arm = &(disassembly->detail->arm);
    int Sd = arm->operands[0].reg;
    int Sn = arm->operands[1].reg;
    int Sm = arm->operands[2].reg;

    if (Sd != ARM_REG_S0 || Sn != ARM_REG_S0 || Sm != ARM_REG_S1 || arm->cc != ARM_CC_AL) return NULL;
    printfdbg("%s %s\n", disassembly->mnemonic, disassembly->op_str);
    printfdbg("This vadd (CC=%d) instruction uses the registers %d, %d, %d %d %d\n", arm->cc, Sd, Sn, Sm, arm->operands[3].reg, arm->operands[4].reg);
    assert(ARM_REG_S0 <= Sd  && Sd <= ARM_REG_S31);
    assert(ARM_REG_S0 <= Sn  && Sn <= ARM_REG_S31);
    assert(ARM_REG_S0 <= Sm  && Sm <= ARM_REG_S31);

    // Make trampoline
    int8_t* tramp = gen_template_tramp(instr_addr);
    if (tramp == NULL) {
        printfdbg("ERROR: failed to generate template trampoline\n");
        exit(1);
    }

    // 'vadd_f32' is connected here as it is the only emulation routine present
    // but in practice you'd want to check the dissassembly above - the variable 'disassembly'.
    link_tramp_to_emu(tramp, &vadd_f32);

    // Put the args (names of S registers) into r0-r2
    tramp_insert_emu_args(tramp, Sd, Sn, Sm, 0);

    printfdbg("Trampoline made for  instruction.");
    return tramp;
}
```

3 fpuemu.h

A stub floating-point emulator which implements virtual registers and an emulation routine for the ‘vadd.f32’ instruction.

```c
#include <math.h>

/*
* 64 single precision registers      = s0 to s63
*                                    = d0 to d31
* This is twice that needed by my test hardware (VFPv3-D16).
*/
#define NUM_SINGLE_PREC_REGS 64
int32_t fpu_registers[NUM_SINGLE_PREC_REGS];

/*
* Converts a single precision register number,
* such as '5' from the register 'r5', to a pointer into memory
* of where the value for that register is stored.
*/
int32_t* sreg_to_bank_ptr(arm_reg reg) {
    assert(ARM_REG_S0 <= reg && reg <= ARM_REG_S31);
    return (int32_t*)fpu_registers + (reg - ARM_REG_S0);
}

// Sets a emulated floating-point register to the given 32-bit value
void set_sreg(arm_reg reg, int32_t val) {
    *sreg_to_bank_ptr(reg) = val;
}

// Returns a value stored in an emulated floating-point register
int32_t get_sreg(arm_reg reg) {
    return *sreg_to_bank_ptr(reg);
}

// Initialise emulator by setting the emulated registers to zero.
void emulator_init() {
    memset(fpu_registers, 0, NUM_SINGLE_PREC_REGS * sizeof(int32_t));
}

/*
* Example of an emulation routine called by a trampoline.
* This is the emulation routine for the 'vadd.f32' instruction and will
* be called from a 'vadd.f32' trampoline. No C code calls this method as
* the machine code in the trampolines are generated at run-time.
*/
void vadd_f32(int32_t Sd, int32_t Sn, int32_t Sm) {
    float a, b;

    /*
    * Copy register values into local variables
    * which are either on the stack or in scratch registers
    * that are restored by the trampoline after this method returns.
    */
    int32_t Sn_val = get_sreg(Sn);
    int32_t Sm_val = get_sreg(Sm);
    memcpy(&a, &Sn_val, sizeof(int32_t));
    memcpy(&b, &Sm_val, sizeof(int32_t));

    // Perform the addition
    float c = a + b;
    int32_t result;

    // Store the result back in a register
    memcpy(&result, &c, sizeof(int32_t));
    set_sreg(Sd, result);

    printfdbg("vadd.f32 Sd:%d Sn:%d Sm:%d: %f + %f = %f\n", Sd, Sn, Sm, a, b, c);
    return;
}
```

4 rmaps.h

At the very beginning of instrumentation a file is opened to retrieve information about the memory layout. This opens, closes, and reads this file by interfacing with the ‘librunt’ library. Librunt’s callback here does the instrumentation for each line in the file.

```c
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <link.h>
#include "maps.h"
#include "dso-meta.h"
#include "librunt.h"
#include "relf.h"

int MAPS_MAX_NUM_LINEBUFS = 100;
int MAPS_BUF_SIZE = 1000;
FILE* maps_file;

/*
* In full transparency, this method resembles that of one of my supervisor's Dr. Stephen Kell.
* His project "libsystrap" uses his other project "librunt" which is also a dependency for this project
* Due to the rigid intended use of the librunt functions "get_a_line_from_maps_fd"
* and "process_one_maps_entry", it was difficult to find a less similar way to implement this.
* Here is a link to that method "trap_all_mappings":
* https://github.com/stephenrkell/libsystrap/blob/790cf958157520ce44afab0bcc2b0fcda9d168fe/example/trace-syscalls.c#L76
*/
static void process_all_lines(int fd, void* callback) {
    typedef char* linebuf_t;
    linebuf_t linebufs[MAPS_MAX_NUM_LINEBUFS];

    /* Allocate line buffers and
    *  Read lines from "/proc/<self>/maps" into the line buffers
    */
    int line_counter = 0;
    int newline_pos;
    while (1) {
        assert(line_counter < MAPS_MAX_NUM_LINEBUFS);

        linebufs[line_counter] = malloc(MAPS_BUF_SIZE);
        newline_pos = get_a_line_from_maps_fd(linebufs[line_counter], MAPS_BUF_SIZE, fd);
        if (newline_pos == -1) break;
        linebufs[line_counter][newline_pos] = '\0';

        line_counter++;
    }

    int num_lines_read = line_counter;  // to be explicit.

    /* Process line buffers (including instrumenting relevant regions) */
    struct maps_entry mline;
    int num_entries_skipped = 0;
    int was_skipped;
    for (int i = 0; i < num_lines_read; i++) {
        if (0 == strncmp(linebufs[i], "00 ", 3)) {  // if the region is not memory mapped
            was_skipped = 1;
            printfdbg("Maps entry began with \"00 \" meaning it was unmapped. Skipping.\n");
        } else {
            was_skipped = process_one_maps_entry(linebufs[i], &mline, callback, NULL);
        }
        if (was_skipped) num_entries_skipped++;
    }

    /* Clean up */
    for (int i = 0; i < num_lines_read; i++) {
        free(linebufs[i]);
    }
}

static int open_maps() {
    maps_file = fopen("/proc/self/maps", "r");
    int fd;
    if ( maps_file == 0 || (fd = fileno(maps_file)) <= 2 ) {
        printfdbg("Unable to open /proc/self/maps: Invalid file descriptor.\n");
        exit(1);
    }
    printfdbg("/proc/self/maps file descriptor: %d\n", fd);
    return fd;
}

static void close_maps() {
    fclose(maps_file);
}
```

5 debug-print.h

Some methods to print useful information when debugging.

```c
/*
* Uncommenting/commenting this 'define' statement will
* enable or disable detailed console output useful for debugging.
*/
//#define DO_DBG_PRINT

#ifdef DO_DBG_PRINT
  #define printfdbg(...) printf (__VA_ARGS__)
#else
  #define printfdbg(...)
#endif

extern long int etext;
extern long int edata;

// Returns 0 or 1 depending on whether an address is in the range [from, to]
static int is_in_range(long unsigned int from, long unsigned int to, long unsigned int addr) {
    return ((from <= addr) && (addr <= to));
}

// Prints if a memory range contains the .text or .data sections
static int print_if_interesting_addr(long unsigned int from, long unsigned int to) {
    if (is_in_range(from, to, &etext)) {
        printfdbg("Range contains .text\n");
    } else if (is_in_range(from, to, &edata)) {
        printfdbg("Range contains .data\n");
    }
    return 0;
}

// Print human-readable information about a programs section headers
static void print_section_headers(struct file_metadata* meta, ElfW(Shdr)* shdrs, int count) {
    printfdbg("Section headers: \n");
    for (int i = 0; i<count; i++) {
        ElfW(Shdr)* shdr = shdrs + i;
        if (shdr == NULL) {
            printfdbg("\tshdrs[%d] does not exist. %d shdrs reported.\n", i, count);
            break;
        }
        int shstrtab_index = shdr->sh_name;
        char* section_name = meta->shstrtab + shstrtab_index;

        char short_name[100];
        if (strlen(section_name) == 0) strcpy(short_name, "(empty header name)");
        else short_name[0] = '\0';

        if (shdr->sh_addr == 0) {
            printfdbg("\t%s%s @ not loaded or base+0\n", short_name, section_name);
        } else {
            printfdbg("\t%s%s \t@ base+%lx\n", short_name, section_name, shdr->sh_addr);
        }
    }
    return;
}

// Print human-readable information about a programs program headers
static void print_program_headers(struct file_metadata* meta, ElfW(Phdr)* phdrs, int count) {
    printfdbg("Program headers: \n");
    for (int i = 0; i<count; i++) {
        ElfW(Phdr)* phdr = phdrs + i;
        if (phdr == NULL) {
            printfdbg("phdrs[%d] does not exist. %d phdrs reported.\n", i, count);
            break;
        }
        printfdbg("\t");
        switch (phdr->p_type) {
            case PT_NULL:
                printfdbg("PT_NULL - ignore the segment");
                break;
            case PT_LOAD:
                printfdbg("PT_LOAD - loadable segment");
                break;
            case PT_DYNAMIC:
                printfdbg("PT_DYNAMIC - dynamic linking info");
                break;
            case PT_INTERP:
                printfdbg("PT_INTERP - location interpreter");
                break;
            case PT_NOTE:
                printfdbg("PT_NOTE - location of Nhdr's (note headers)");
                break;
            case PT_SHLIB:
                printfdbg("PT_SHLIB - reserved");
                break;
            case PT_PHDR:
                printfdbg("PT_PHDR - location of program header table");
                break;
            case PT_LOPROC:
                printfdbg("PT_LOPROC - reserved");
                break;
            case PT_HIPROC:
                printfdbg("PT_HIPROC - reserved");
                break;
            case PT_GNU_STACK:
                printfdbg("PT_GNU_STACK - used by kernel");
                break;
            default:
                printfdbg("Unknown (%ld)", phdr->p_type);
                break;
        }
        printfdbg("\n");
        if (phdr->p_type == PT_LOAD) { // or any other type but we only care about PT_LOAD
            printfdbg("\t\tp_offset := %x\n", phdr->p_offset);
            printfdbg("\t\tp_vaddr := %x\n", phdr->p_vaddr);
            printfdbg("\t\tp_filesz := %x\n", phdr->p_filesz);
            printfdbg("\t\tp_memsz := %x\n", phdr->p_memsz);
            printfdbg("\t\tp_align := %x\n", phdr->p_align);
        }
    }
    return;
}

// Prints more information about an entry in the '/proc/<pid>/maps' file
static int print_maps_entry(struct maps_entry *ent, char *linebuf, void *arg) {
    struct file_metadata* meta = __runt_files_metadata_by_addr(ent->first);
    if (meta == NULL) {
        printfdbg("Error: could not retrieve metadata for file %s\n", ent->rest);
        return 1;
    }
    ElfW(Ehdr)* ehdr = meta->ehdr;

    printfdbg("%lx to %lx\n", ent->first, ent->second);
    print_if_interesting_addr(ent->first, ent->second);
    printfdbg("Privileges: %c%c%c%c\n", ent->r, ent->w, ent->x, ent->p);
    printfdbg("Inode: %d\n", ent->inode);
    printfdbg("Filename: %s\n", meta->filename);

    printfdbg("Magic: %s\n", ehdr->e_ident);
    printfdbg("Arch: ");
    switch(ehdr->e_machine) {
        case EM_X86_64:
            printfdbg("EM_X86_64");
            break;
        case EM_IA_64:
            printfdbg("EM_IA_64");
            break;
        case EM_ARM:
            printfdbg("EM_ARM");
            break;
        case EM_NONE:
            printfdbg("EM_NONE");
            break;
        case EM_386:
            printfdbg("EM_386");
            break;
        default: printfdbg("Unknown - %d", ehdr->e_machine);
    }
    printfdbg("\n");
    printfdbg("Section header table file offset: %ld\n", ehdr->e_shoff);
    printfdbg("\tentry size: %d\n", ehdr->e_shentsize);
    printfdbg("\tentry count: %d\n", ehdr->e_shnum);
    printfdbg("Section header string table index: %d\n", ehdr->e_shstrndx);
    printfdbg("Segment header table file offset: %ld\n", ehdr->e_phoff);
    printfdbg("\tentry size: %d\n", ehdr->e_phentsize);
    printfdbg("\tentry count: %d\n", ehdr->e_phnum);

    print_section_headers(meta, meta->shdrs, ehdr->e_shnum);
```

```c
    print_program_headers(meta, meta->phdrs, ehdr->e_phnum);

    printfdbg("\n");
    return 0;
}
```