ELM 334 - Homework #3

Ömer Emre Polat
1801022037

A. Problem 1

In 32-bit arm function parameters are passed by pushing the input variables to stack and then when the function runs pop the values back to the data registers and perform the function on them. We could give examples to each problems taking the base function as void func(int a).

For 4 or less parameters the parameters are moved into r0-r3 registers and used in the function.

a. For a single parameter

mov r3, a

bl 0 <func>

…

b. For two parameters

mov r2, a

mov r3, b

bl 0 <func>

…

c. For three parameters

mov r1, a

mov r2, b

mov r3, c

bl 0 <func>

…

d. For four parameters

mov r0, a

mov r1, b

mov r2, c

mov r3, d

bl 0 <func>

…

For more than four parameters the r3 is used to push the extra values to stack and then popping them back when they are needed in the function.

e. For five parameters

```
mov r3, e

adds sp, sp, #4

str r3 [sp]

mov r0, a

mov r1, b

mov r2, c

mov r3, d

bl 0 <func>

subs sp, sp, #4

…
```

f. For six parameters

```
mov r3, f

adds sp, sp, #4

str r3, [sp]

mov r3, e

adds sp, sp, #4

str r3, [sp]

mov r0, a

mov r1, b

mov r2, c

mov r3, d

bl 0 <func>

subs sp, sp, #8

…
```

B. Problem 2

When a function has a return value in assembly, the return value is stored in the data register eax for use in other parts of the code. Then the “ret” command returns the value stored in eax for the return operation.

```
int func(int a){ … return x; }

mov eax, x

ret
```

C. Problem 3

After downloading the .elf file we can use GNU ARM Toolchain tool arm-none-eabi-objdump -D command on it to disassemble.

Tools	Generic command name	Command name in GNU Tools for ARM® embedded processors
C Compiler	gcc	arm-none-eabi-gcc
Assembler	as	arm-none-eabi-as
Linker	ld	arm-none-eabi-ld
Binary file generation tool	objcopy	arm-none-eabi-objcopy
Disassembler	objdump	arm-none-eabi-objdump

```
C:\Users\user>arm-none-eabi-objdump -D Desktop\2020-hw3.elf

Desktop\2020-hw3.elf:     file format elf32-littlearm
```

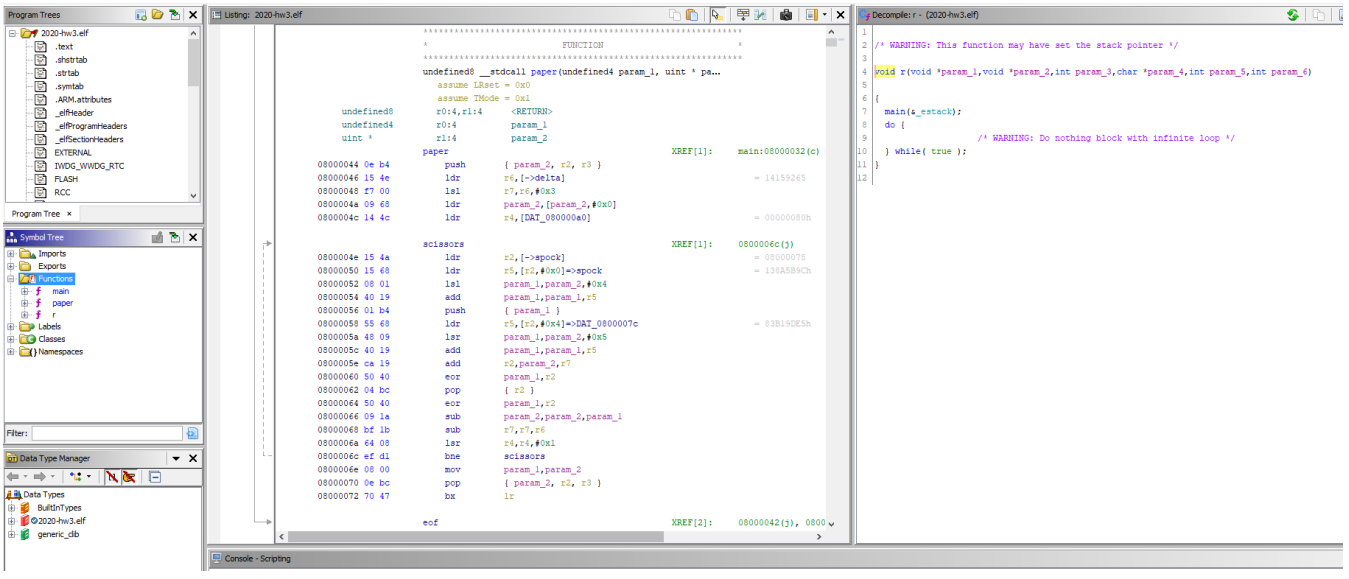
After the code is disassembled we can comment on each of the commands to keep track of them

```
Disassembly of section .text:

08000000 <v>:
 8000000:       10002000        andne   r2, r0, r0 ; moves r0 to r2 if not equal
 8000004:       08000021        stmdaeq r0, {r0, r5} ; ?
 8000008:       0800002b        stmdaeq r0, {r0, r1, r3, r5} ; store multiple decrement after equal
 800000c:       0800002b        stmdaeq r0, {r0, r1, r3, r5} ; duplicate command
 8000010:       10000000        andne   r0, r0, r0 ; ?
 8000014:       10000000        andne   r0, r0, r0 ; ?
 8000018:       10000000        andne   r0, r0, r0 ; ?
 800001c:       10000000        andne   r0, r0, r0 ; ?

08000020 <r>:
 8000020:       481b            ldr     r0, [pc, #108]  ; (8000090 <lizard+0x10>) loads 8000090 to r0
 8000022:       4685            mov     sp, r0 ; moves r0(8000090) to stack pointer
 8000024:       f000 f802       bl      800002c <main> ; branch with link to main
 8000028:       e7fe            b.n     8000028 <r+0x8> ; indefinite loop

0800002a <d>:
 800002a:       e7fe            b.n     800002a <d> ; indefinite loop

0800002c <main>:
 800002c:       4919            ldr     r1, [pc, #100]  ; (8000094 <lizard+0x14>) loads 8000094 to r1
 800002e:       4a1a            ldr     r2, [pc, #104]  ; (8000098 <lizard+0x18>) loads 8000098 to r2
 8000030:       2300            movs    r3, #0 ; moves 0 to r3

08000032 <rock>:
 8000032:       f000 f807       bl      8000044 <paper> ; branch link to paper
 8000036:       6010            str     r0, [r2, #0] ; store the value that r2 points to r0
 8000038:       3104            adds    r1, #4 ; add 4 to r1 (pointer arithmetic)
 800003a:       3204            adds    r2, #4 ; add 4 to r2 (pointer arithmetic)
 800003c:       3301            adds    r3, #1 ; add 4 to r3 (pointer arithmetic)
 800003e:       2b04            cmp     r3, #4 ; compare and update flags (r3-4)
 8000040:       d1f7            bne.n   8000032 <rock> ; if not equal branch to rock
 8000042:       e017            b.n     8000074 <eof> ; indefinite loop

08000044 <paper>:
 8000044:       b40e            push    {r1, r2, r3} ; push r1, r2 and r3 to stack
 8000046:       4e15            ldr     r6, [pc, #84]   ; (800009c <lizard+0x1c>) ; load to r6 *(800009c)
 8000048:       00f7            lsls    r7, r6, #3 ; logical shift left (r7 = r6 << 3)
 800004a:       6809            ldr     r1, [r1, #0] ; dereference r1
 800004c:       4c14            ldr     r4, [pc, #80]   ; (80000a0 <lizard+0x20>) ; move *(80000a0) to r4

0800004e <scissors>:
 800004e:       4a15            ldr     r2, [pc, #84]   ; (80000a4 <lizard+0x24>) ; move *(80000a4) to r2
 8000050:       6815            ldr     r5, [r2, #0] ; load *(r2) to r5
 8000052:       0108            lsls    r0, r1, #4 ; logical shift left (r0 = r1 << 4)
 8000054:       1940            adds    r0, r0, r5 ; add r65 to r0
 8000056:       b401            push    {r0} ; push r0 to stack
 8000058:       6855            ldr     r5, [r2, #4] ; load *(r2+1) to r5
```

```
 800005a:       0948            lsrs    r0, r1, #5 ; logical shift right (r0 = r1 >> 5)
 800005c:       1940            adds    r0, r0, r5 ; add r5 to r0
 800005e:       19ca            adds    r2, r1, r7 ; add r1 to r7 and write to r2
 8000060:       4050            eors    r0, r2 ; xor r0 and r2 then write to r0
 8000062:       bc04            pop     {r2} ; push r2 to stack
 8000064:       4050            eors    r0, r2 ; xor r0 and r2 then write to r0
 8000066:       1a09            subs    r1, r1, r0 ; r1 = r1 - r0
 8000068:       1bbf            subs    r7, r7, r6 ; r7 = r7 - r6
 800006a:       0864            lsrs    r4, r4, #1 ; logical shift right (r4 = r4 >> 1)
 800006c:       d1ef            bne.n   800004e <scissors> ; if not equal branch to scissors
 800006e:       0008            movs    r0, r1 ; move r0 to r1
 8000070:       bc0e            pop     {r1, r2, r3} ; pop back 3 values from stack
 8000072:       4770            bx      lr ; bx lr to switch the state to arm from thumb

08000074 <eof>:
 8000074:       e7fe            b.n     8000074 <eof> ; indefinite loop
 8000076:       46c0            nop                     ; (mov r8, r8)

08000078 <spock>:
 8000078:       138a5b9c        orrne   r5, sl, #156, 22        ; 0x27000
 800007c:       83b19de5                        ; <UNDEFINED> instruction: 0x83b19de5

08000080 <lizard>:
 8000080:       a2390c55        eorsge  r0, r9, #21760  ; 0x5500
 8000084:       113f39fc        teqne   pc, ip  ; <illegal shifter operand>     ; <UNPREDICTABLE>
 8000088:       6140f4fd        strdvs  pc, [r0, #-77]  ; 0xfffffffb
 800008c:       d3926c34        orrsle  r6, r2, #52, 24 ; 0x3400
 8000090:       10002000        andne   r2, r0, r0 ; if not equal move r0 to r2
 8000094:       08000080        stmdaeq r0, {r7} ; store multiple decrement address r0 to r7
 8000098:       10000200        andne   r0, r0, r0, lsl #4
 800009c:       14159265        ldrne   r9, [r5], #-613 ; 0xfffffd9b
 80000a0:       00000080        andeq   r0, r0, r0, lsl #1
 80000a4:       08000078        stmdaeq r0, {r3, r4, r5, r6} ; store multiple decrement address r0 r3-r6

Disassembly of section .ARM.attributes:

00000000 <.ARM.attributes>:
   0:   00002141        andeq   r2, r0, r1, asr #2
   4:   61656100        cmnvs   r5, r0, lsl #2
   8:   01006962        tsteq   r0, r2, ror #18
   c:   00000017        andeq   r0, r0, r7, lsl r0
  10:   726f4305        rsbvc   r4, pc, #335544320      ; 0x14000000
  14:   2d786574        cfldr64cs       mvdx6, [r8, #-464]!     ; 0xfffffe30
  18:   002b304d        eoreq   r3, fp, sp, asr #32
  1c:   4d070c06        stcmi   12, cr0, [r7, #-24]     ; 0xfffffff8
  20:   Address 0x00000020 is out of bounds.
```

The code is commented on what it does. I've tried to follow along the code but some parts were too complicated and unidentified instruction and illegal shift operands were confusing.

I've tried using other software to disassemble and decompile the code back to c. I've found two good candidates for software IDA and Ghidra. I've chosen Ghidra since it had a SVD memory map loader script which might make the job easier.



Even then there were problems with missing data and parts which made the decompiled code very hard to follow.

Even then I've managed to extract three functions which are given below.

main.c

```
1
2  void main(undefined4 param_1)
3
4  {
5    uint *puVar1;
6    undefined4 *puVar2;
7    int Counter;
8    undefined8 uVar3;
9
10   puVar1 = (uint *)&lizard;
11   puVar2 = (undefined4 *)&serial;
12   Counter = 0;
13   do {
14     uVar3 = paper(param_1,puVar1);
15     param_1 = (undefined4)((ulonglong)uVar3 >> 0x20);
16     *puVar2 = param_1;
17     puVar1 = (uint *)((int)uVar3 + 4);
18     puVar2 = puVar2 + 1;
19     Counter = Counter + 1;
20   } while (Counter != 4);
21   do {
22                     /* WARNING: Do nothing block with infinite loop */
23   } while( true );
24 }
25
```

paper.c

```
1
2  undefined8 paper(undefined4 param_1,uint *param_2)
3
4  {
5    uint uVar1;
6    uint uVar2;
7    int iVar3;
8
9    iVar3 = -0x5f536cd8;
10   uVar1 = *param_2;
11   uVar2 = 0x80;
12   do {
13     uVar1 = uVar1 - ((uVar1 >> 5) + 0x83b19de5 ^ uVar1 + iVar3 ^ uVar1 * 0x10 + 0x138a5b9c);
14     iVar3 = iVar3 + -0x14159265;
15     uVar2 = uVar2 >> 1;
16   } while (uVar2 != 0);
17   return CONCAT44(uVar1,param_2);
18 }
19
```

r.c

```
1
2  /* WARNING: This function may have set the stack pointer */
3
4  void r(void *param_1,void *param_2,int param_3,char *param_4,int param_5,int param_6)
5
6  {
7    main(&_estack);
8    do {
9                        /* WARNING: Do nothing block with infinite loop */
10   } while( true );
11 }
12
```

There were variables named after rock paper scissor spock lizard game which is a 5 choice versio of the rock paper scissors game.

D. Problem 4

Dangers and Methods of Fault Injections

In order to extract information about a cryptosystem or exploit some workings of a cryptosystem, an attacker can induce specific faults and errors to the system. Fault attacks are specifically more stronger against unprotected systems that any other attack types. But what types of fault attacks exist and how do fault injections can be performed.

For injection methods, there are many ways for it. The most common ones include the glitch attack, temperature attack, light attack and magnetic attack.

Glitch attack can be done by inducing a certain glitch to the system by variations on the power supply or an external clock manipulation. In order for this technique to succeed the attacker has to control the variations on the power supply unit. This technique is one of the most common techniques for breaking into several cryptosystems. Reasons as to why this is most common could be that it is easy to apply because the attacker doesn't have to know about the localizations of the system. For temperature attack, it works exactly as it sounds. The temperature of certain part of the system is altered into the limit of where they start to not work properly. Random modifications of RAM cells or read and write mismatches can occur in the results of this attack. Most advanced cryptosystems use a temperature sensor but still a mismatch can occur between the memory's operating temperature and the sensors detecting range. For light attacks, it is one of the most accurate techniques out there since it uses the photoelectric phenomenon. This attack type can be very powerful against many systems since most of the are still exposed to light interference. To succeed in this technique attacker must control the light beams general position, wavelength, energy and the duration. Because there are many variables that an attacker must control the light used is mostly a laser beam. For magnetic attacks, magnetic waves can be used to induce a current to certain data paths to create faults in the system. This attack can be performed with cheap tools but it lacks the accuracy and precision of a light attack performed with a laser.

In choosing a method to attack, the attacker must know what types of fault are going to be created in the system. There are two types of faults. One is the transient and the other one is the permanent. Transient faults are not permanent and the system can recover after a reset or when the attack ceases. Still for all these attacks the attacker must know what errors to induce into the system in order to get the desired result. These desired results many be a bit or a byte change induced to a system. Usually its more possible to induce a byte change to the system rather than a bit change in current technology systems. The attacked can desire to change a specific data value to some other value or a random value. Generally it is easier to induce a random value to this system. The attacker can desire to create faults on the computations to create faults on the memory of the system. Computational errors are easy to induce but changing a value in memory may be a lot difficult. Lastly an attacked can desire a control error to skip some iterations or operation on the system to create desired faults. Although this technique is difficult it can prove to be very powerful.

E. Problem 5

Controlling the Program Counter on ARM with Fault Injections

ARM or MIPS systems are designed around a single system on chip which usually houses one or more central processing units. Software running on most embedded systems are vulnerable and exploitable. An open research shows that there are roughly 0.434 defects per thousand lines of code. There may still be no vulnerabilities on shorter codes but there are other techniques for fault injections on such systems aswell. These techniques include clock fault injection, voltage fault injection electromagnetic injection and optical fault injection. Using these techniques the injected fault in theory can have an impact on the stages of the instruction cycle. It is typically unknown what exactly goes wrong within the CPU when a fault is injected but it is easier to observe the modified behavior itself. Two modified behaviors are possible: instruction corruption or instruction skipping. Instruction corruption is corrupting the instruction to another wanted instruction. Instruction skipping is just a subset of instruction corruption and includes skipping an instruction in order to inject faults at the system. On ARM system most efficient load and store operation are performed using LDMIA and STMIA. From an attackers standpoint these instructions are a main target for fault injection attacks.

In most other architectures program counter cannot be modified by direct instructions. LDMIA is especially a target for injection attacks because performing a successful attack on an LDMIA instruction is higher than a LDR instruction. For example we will take two attack scenarios: a boot attack and a runtime attack. This boot attack is performed during the initialization of the embedded system. Although a SOC includes multiple memories like ROM, RAM or Flash, it usually has external memory units aswell. SOC cannot trust these external memory units since these units are accessible from other external units. Thus the usual procedure to take is to store the code to external memory, copy that code to internal memory and then perform a cryptographic verification. Because of the nature of the booting, an attacker may choose to do the attack on a boot stage if he wants to execute a high privileged code. In the other scenario we can assume a runtime attack on the trusted execution environment. For this type of attack there are certain problems that an attacker has to solve. First an attacker would need to identify an API command into TEE. Second the data coming from the non-secure context is under control of the attacker, assuming all control of REE is gained. Therefore the attack has full control over what passes to the API. Third the data going to API must be constructed. Finally the fault is injected when the payload is placed into place.

For simulating these events we can use simulation programs. These simulations help us determine the likelihood of a fault injection event occurring in a system. The simulation program is run natively on an ARM board to ensure the simulation accuracy. Using a technique we test 41448 different instructions derived from the original instruction. A bit flipper code is used in this simulation. The simulation program loads the data using LDR instructions and loads the controlled value into the program counter register. When R3 register is used there are multiple corruptions on instructions when compared to only a single corruption that happens when we use R7. For another simulation we use LDMIA instruction where the address is stored in the R0 and the values are loaded into R4, R5, R6 and R7 registers. This simulation shows that there are more modified or corrupted instruction on the LDMIA test which can state that there could’ve been a successful glitch may change the operands.

For experimental results we could use the same platform used on the simulation before but add some control over the board. These controls include: being able to perform power cuts on the PCB, removing the stabilizing capacitors from the board, adding a trigger for the general purpose IO and the reset signal being fed into the fault injection system. For the fault injection setup we use a high speed FPGA glitcher connected to the target from the power, reset and trigger. The test applications used for the experiments are implemented as an additional command in U-Boot, which is the de facto standard boot loader embedded systems and is publicly available as open source software.

After the both LDR and LDMIA experiments we get certain results. These results show that LDR experiment out of 10000 tests, only a %0.01 of the tests had a successful glitch while %34.36 were mutes or resets and the rest being no effect. When done with LDMIA instruction the 10000 tests show that %0.27 of the tests were a successful glitch while %26.53 mutes or resets and rest of them having no effect on the system. This shows that the more efficient LDMIA instruction is more vulnerable to certain attacks against fault injection. This difference is mostly caused by the difference in instruction encoding of both instructions.

The effective countermeasures we can have against fault injection include deflection. Manipulation of a system usually requires the correct timing to inject the faults. The system can be run on a varying speed so that the attacker can’t get an easy timing correction on fault injections. Another countermeasure technique could be detecting the faults. If faults can be timely detected it is possible to ignore them and prevent unintended behavior. Last countermeasure could be to react to the injection attempts. This reaction could include imposing penalties when a fault injection is detected.

In conclusion ARM32 (AArch32) architecture could get fault injected and manipulated at the program counter by external attackers. The likeliness of an attack being successful is determined by the instructions used in the main code of the system. Taking countermeasures against these injection methods are necessary to create a safer systems.