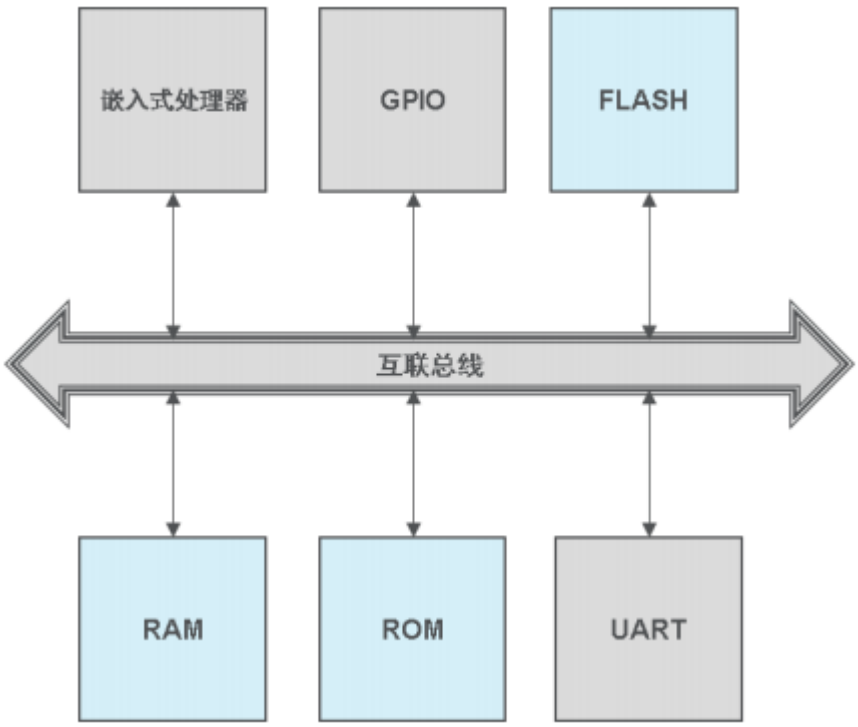
嵌入式安全实验_固件篡改和完整性度量

1. 实验环境

- 软件部分
  - boot和FW开发平台

    ```
    Linux adminme-virtual-machine 6.5.0-28-generic #29~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC T
    ```

  - 基于课程实验给出的Demo继续开发
  - 软件编译环境：Opentitan环境（主要是其toolchain），参考 https://opentitan.org/book/doc/getting_started/index.html
- 硬件部分
  - 仿真平台
    - 操作系统：Windows 10 Pro 10.0.18363
    - 仿真软件：ModelSim 2020.02
- 整体结构

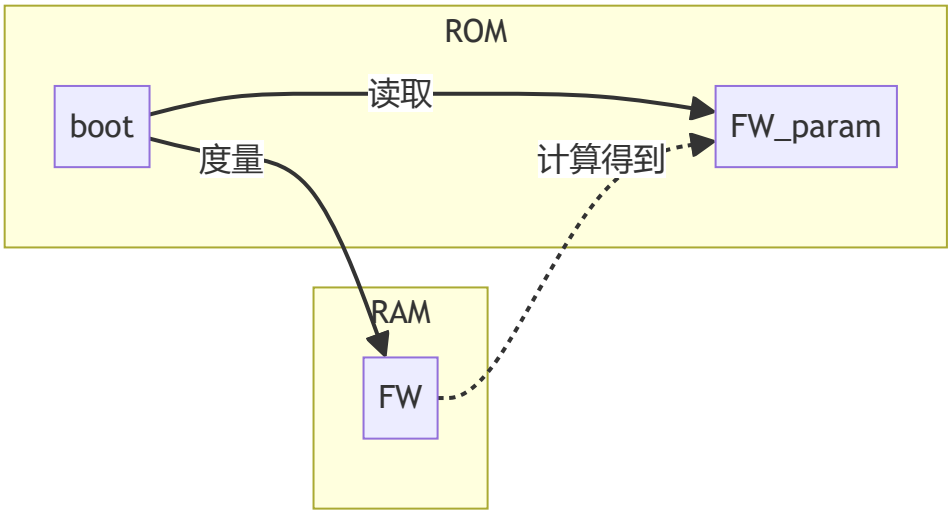


2. 实验目标

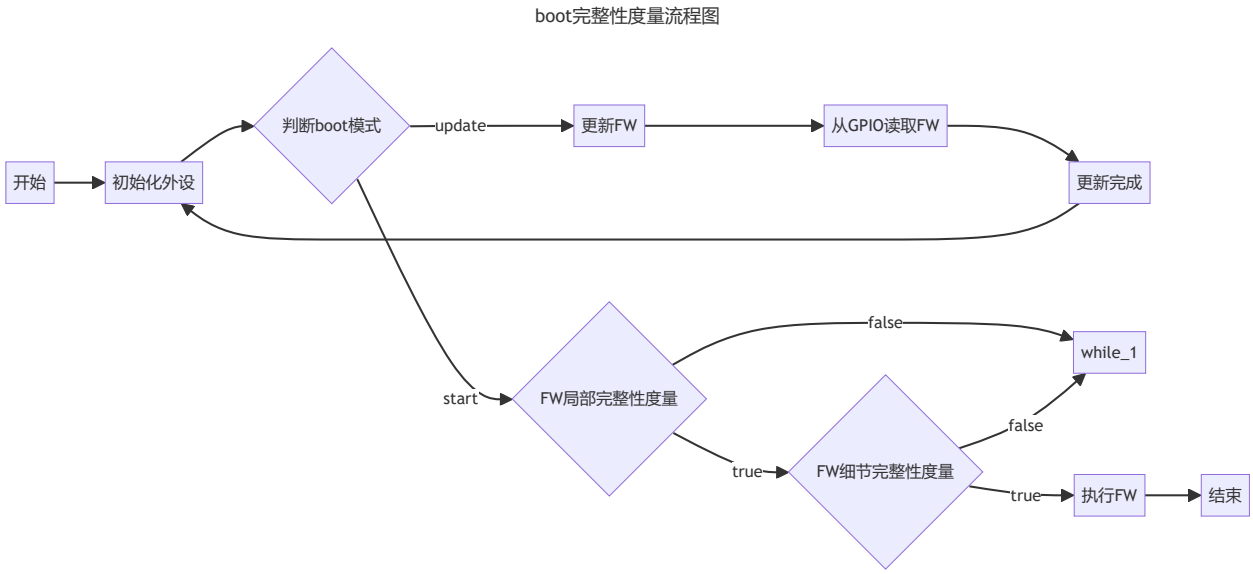
- boot和FW正常运行，boot正常启动FW
- boot对FW进行完整性度量（可使用XOR、CRC、HASH等）
- FW实现一些简单的功能（例如打印字符串、GPIO控制等）
- 拓展
  - boot实现update FW功能，通过GPIO获取更新的FW，并覆盖FW
  - 利用boot的更新FW功能，模拟通过GPIO写入被篡改的FW，通过boot的完整性度量阻止FW运行

3. 实验设计

- 整体设计



- 完整性度量设计
  - 计算FW的完整性相关参数，将FW参数写入到ROM中，由boot读取参数来度量FW的完整性
  - 设计通过两次度量来验证的完整性
    - 完整性参数：FW的长度，FW的局部完整性参数，FW的细节完整性参数
    - 第一次：读取FW的长度，通过XOR计算FW的第一个、中间的(len>>1)、最后一个4字节，按照first^{center}last顺序计算得到**局部完整性参数**。检测较为明显的恶意FW。
    - 第二次：读取FW全部数据，计算CRC32的值，得到**细节完整性参数**。检测考虑到FW的每个字节


boot完整性度量流程图
开始
初始化外设
判断boot模式
update
更新FW
从GPIO读取FW
更新完成
start
FW局部完整性度量
false
while_1
true
FW细节完整性度量
false
true
执行FW
结束

- boot中部分代码

```
void _boot_start(void)
{
    boot_stat_t boot_stat = BOOT_START_MODE;
    uint32_t update_data = 0;
    #ifdef CHK_FW
    check_status_t ret_val = CHECK_NOT_PASS;
    #endif

    uart_configure(115200);

    print_str("\r");
    print_str("Embedded Security BootLoader Demo...\r\n");

    boot_stat = get_boot_mode();

    if(boot_stat == BOOT_UPDATE_FW_MODE) {
        // Note: 只读一个字节，让将一个字节写入FW的首字节
        // 主要是为了测试通过boot update FW后做完整性校验
        // 是被恶意写入的FW无法运行

        // gpio pin1 ACK
        REG32(GPIO_0_OE) |= 1<<1;
        WAIT_THREE_CYCLE;
        REG32(GPIO_0_OUT) |= 1<<1;
        WAIT_THREE_CYCLE;
        print_str("update FW ...\r\n");

        // update FW
        update_data = REG32(GPIO_0_IN);
        GET_FW_FIRST_32_B = update_data;
        WAIT_THREE_CYCLE;

        // reboot
        print_str("\rupdate ok, reboot...\r\n");
        __asm__ volatile ("jal zero, 0x00008080");
    }

    #ifdef CHK_FW
    // local
    ret_val = fw_check_local_param();
    if(CHECK_NOT_PASS == ret_val) goto lb_boot_fail;

    // detail
```

```
    ret_val = CHECK_NOT_PASS;
    ret_val = fw_check_detail_param();
    if(CHECK_NOT_PASS == ret_val) goto lb_boot_fail;
    #endif

    _flash_header.entry();

    #ifdef CHK_FW
    lb_boot_fail:
        BOOT_FAIL;
    #endif
}

boot_stat_t get_boot_mode(void)
{
    uint32_t gpio_0_in = REG32(GPIO_0_IN);

    /// gpio0 pin0 输入高电平 则进入更新固件
    if((gpio_0_in&(0x00000001)) == 1) return BOOT_UPDATE_FW_MODE;

    return BOOT_START_MODE;
}
```

- 完整性参数如何写入ROM
  - 构建一个脚本工具，功能如下
    - 编写好FW源码后，make后得到FW的bin，脚本计算FW的长度、局部完整性参数、细节完整性参数
    - 自动生成 `sec_init.S` ，和boot一起编译生成boot的镜像，由 `sec_init.S` 向rom中写入完整性参数

```
.section .sec_param, "a"
/* 保留 */
.org 0x0
.long 0x00000000
.org 0x4
.long 0x00000000
.org 0x8
.long 0x00000000

/* FW 细节完整性参数 */
.org 0xc
.long 0xd5496fda

/* FW 局部完整性参数 */
.org 0x10
.long 0x224ab9b7

/* FW 长度 */
.org 0x14
.long 0x3f6
```

- 然后再编译boot，得到boot的bin，同时由于添加了新的.section，需要修改boot的link脚本，在SECTION中添加新的section sec_param，位置放中断向量表后(vectors)

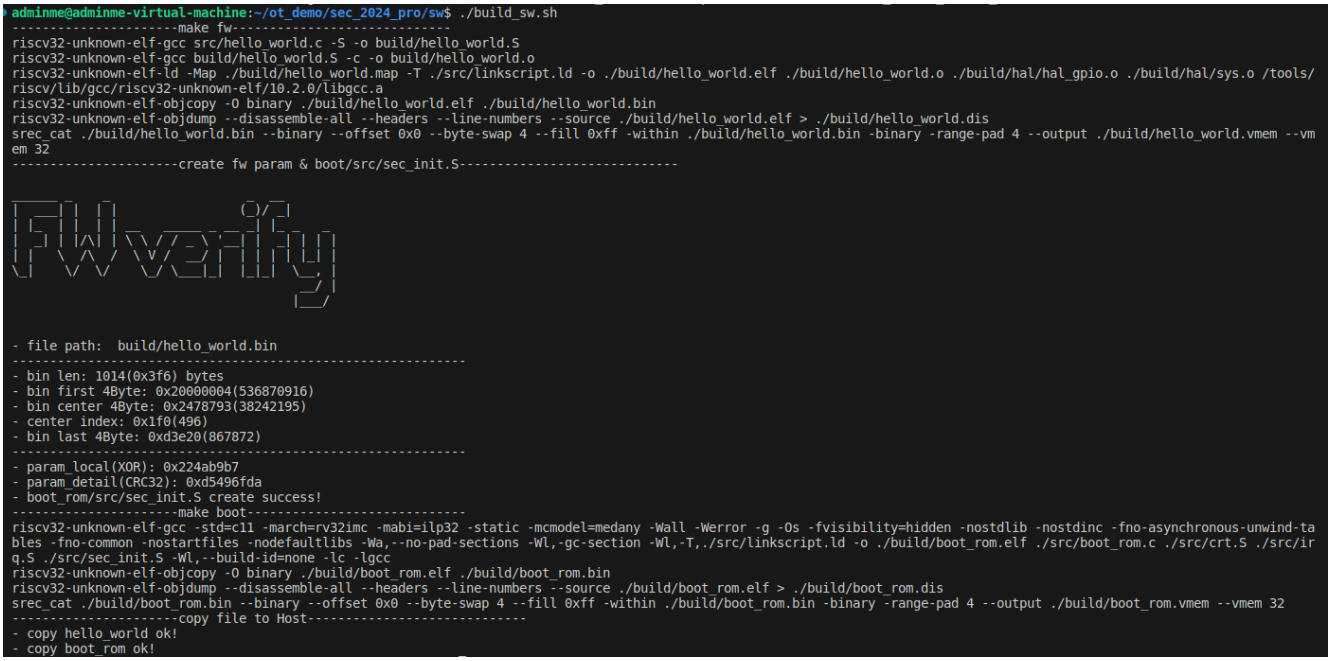
```
SECTIONS {
    .vectors _boot_address : ALIGN(4) {
        KEEP(*(.vectors))
        *(.vectors)
    } > rom

    .sec_param : ALIGN(4) {
        KEEP(*(.sec_param))
        *(.sec_param)
    } > rom
...
```

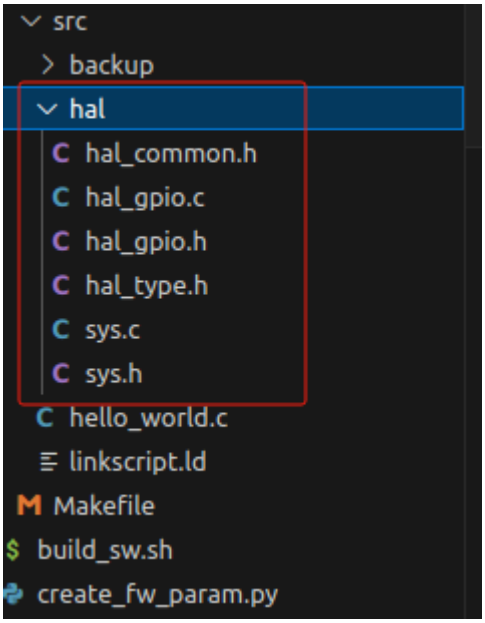
- 执行效果，使用脚本 `build_sw.sh` 执行编译和脚本

```
#! /bin/bash

echo '----------------------make fw-----------------------------' &&
cd hello_world/ && make all &&
echo '----------------------create fw param & boot/src/sec_init.S-----------------
../create_fw_param.py build/hello_world.bin
echo '----------------------make boot---------------------------' &&
cd ../boot_rom/ && make all && cd ../ &&
echo '----------------------copy file to Host-------------------------' &&
cp ./hello_world/build/hello_world.vmem /mnt/hgfs/VM_WIN_GMEM/ && echo "- copy hell
cp ./boot_rom/build/boot_rom.vmem /mnt/hgfs/VM_WIN_GMEM/ && echo "- copy boot_rom o
```



- FW功能设计
  - FW保证正常运行即可，为了便于编码，添加了一层HAL（硬件抽象层）



- 修改了FW原本项目结构，就需要修改Makefile，修改后的如下

```
CCARGS = -std=c11 -march=rv32imc -mabi=ilp32 -static -mcmodel=medany -Wall -Werror

CC = riscv32-unknown-elf-gcc
LD = riscv32-unknown-elf-ld
OC = riscv32-unknown-elf-objcopy
OD = riscv32-unknown-elf-objdump
HEX = srec_cat

FW_NAME = hello_world
FW_TOP_DIR = ./src
FW_HAL_DIR = ./src/hal

FW_BUILD_DIR = ./build

FW_LINK_SCRIPT = ${FW_TOP_DIR}/linkscript.ld

HAL_SRC = $(wildcard ${FW_HAL_DIR}/*.c)
HAL_ASM = $(patsubst ./src/%.c,./build/%.S,${HAL_SRC})
HAL_OBJ = $(patsubst %.S,%.o,${HAL_ASM})

FW_SRC = ${FW_TOP_DIR}/${FW_NAME}.c
FW_ASM = ${FW_BUILD_DIR}/${FW_NAME}.S
FW_OBJ = ${FW_BUILD_DIR}/${FW_NAME}.o
FW_ELF = ${FW_BUILD_DIR}/${FW_NAME}.elf
FW_MAP = ${FW_BUILD_DIR}/${FW_NAME}.map
FW_VMEM = ${FW_BUILD_DIR}/${FW_NAME}.vmem
FW_BIN = ${FW_BUILD_DIR}/${FW_NAME}.bin
FW_DIS = ${FW_BUILD_DIR}/${FW_NAME}.dis

LIB = /tools/riscv/lib/gcc/riscv32-unknown-elf/10.2.0/libgcc.a

# 编译
${HAL_ASM}:${HAL_SRC}
    $(CC) -S ${HAL_SRC}
    for file in *.s; do \
        mv "$$file" "${FW_BUILD_DIR}/hal/$${file%.s}.S"; \
    done

$(FW_ASM): $(FW_SRC)
    $(CC) $< -S -o $@

# 汇编
${HAL_OBJ}: ${HAL_ASM}
```

```
    $(CC) -c ${HAL_ASM}
    mv ./*.o ${FW_BUILD_DIR}/hal/
$(FW_OBJ): $(FW_ASM)
    $(CC) $< -c -o $@

# 链接
link: $(HAL_OBJ) $(FW_OBJ)
    ${LD} -Map ${FW_MAP} -T ${FW_LINK_SCRIPT} -o ${FW_ELF} $(FW_OBJ) $(HAL_OBJ) ${L

all: link
    ${OC} -O binary ${FW_ELF} ${FW_BIN}
    ${OD} --disassemble-all --headers --line-numbers --source ${FW_ELF} > ${FW_DIS}
    ${HEX} ${FW_BIN} --binary --offset 0x0 --byte-swap 4 --fill 0xff -within ${FW_B

clean:
    rm ${FW_ASM} ${FW_OBJ} ${FW_ELF} ${FW_MAP} ${FW_VMEM} ${FW_BIN} ${FW_DIS}  -f
    rm ${FW_BUILD_DIR}/hal/*.S -f
    rm ${FW_BUILD_DIR}/hal/*.o -f
```

- FW实现的简单功能
  - 打印字符"Hi UCAS"
  - 初始化的GPIO_0 Pin12和Pin31
  - 死循环执行：Pin12输出高电平,Pin31输出低电平，延时8us后，Pin12和Pin31电平反转，再延时24us
- FW部分代码

```c
int main(void)
{
    print_str(
        " _   _ _____   _   _ _____   ___   _____\r"
        "| | | |_   _| | | | /  __ \\ / _ \\ /  ___|\r"
        "| |_| | | |   | | | | /  \\// /_\\ \\ `--. \r"
        "|  _  | | |   | | | | |    |  _  | `--. \\\r"
        "| | | |_| |_  | |_| | \\__/\\| | | |/\\__/ /\r"
        "\\_| |_/\\___/   \\___/ \\____/\\_| |_/\\____/ \r"
    );
    init_gpio_0(GPIO_PIN_n(12)|GPIO_PIN_n(31));
    print_str("[init] gpio ok >\r");
    while (1)
    {
        set_gpio_0(GPIO_PIN_n(12));
        reset_gpio_0(GPIO_PIN_n(31));
        delay_us(8);

        reset_gpio_0(GPIO_PIN_n(12));
        set_gpio_0(GPIO_PIN_n(31));
        delay_us(24);
    }
    return 0;
}
```

4. 实验测试

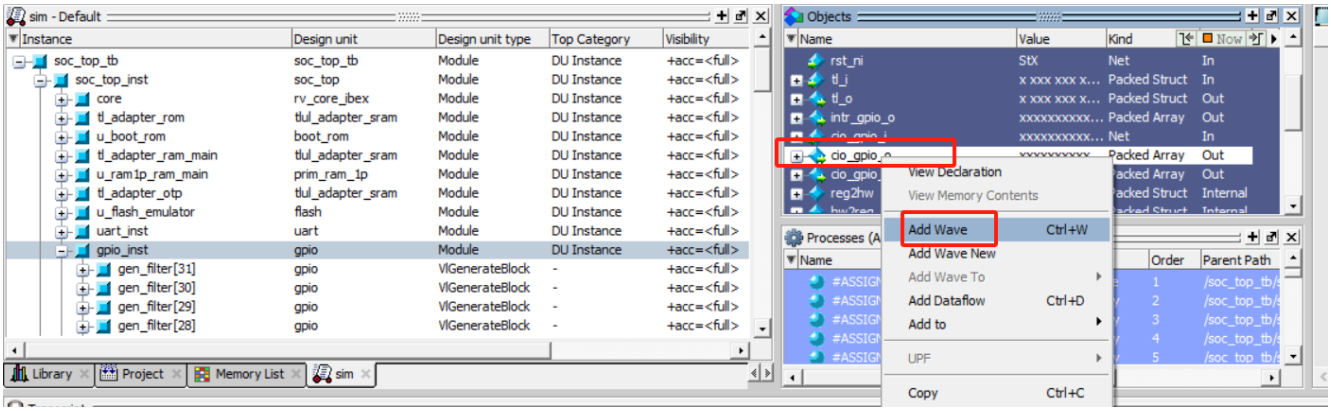
- FW完整性度量测试（建立ModelSim工程步骤略），使用项目默认的 `soc_top_tb.v` 作为激励文件（Testbench）
  - 准备了3个FW，1个为正式版本，其他2个都被篡改了

```
# 正式版本，没有修改的
# print_str("[init] gpio ok\r");
- param_local(XOR): 0x224aecf8
- param_detail(CRC32): 0x735f9f5f

# 修改的FW，通过XOR可以检测的
# 修改为 print_str("[init] gpio ok >\r");
- param_local(XOR): 0x224ab9b7
- param_detail(CRC32): 0xd5496fda

# 修改的FW，通过CRC32可以检测的
# 修改为 print_str(">init] gpio ok\r");
- param_local(XOR): 0x224aecf8
- param_detail(CRC32): 0x3047cfce
```

- 添加 `cio_gpio_o` 输出到Wave，为了FW正常启动后输出的波形

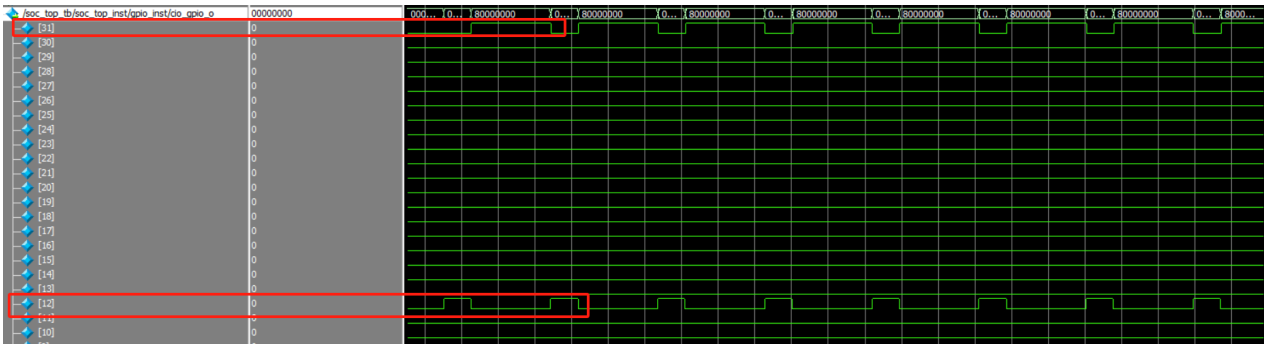


- 启动仿真，等待运行结果（串口打印、波形输出）
  - 串口输出，通过了完整性校验

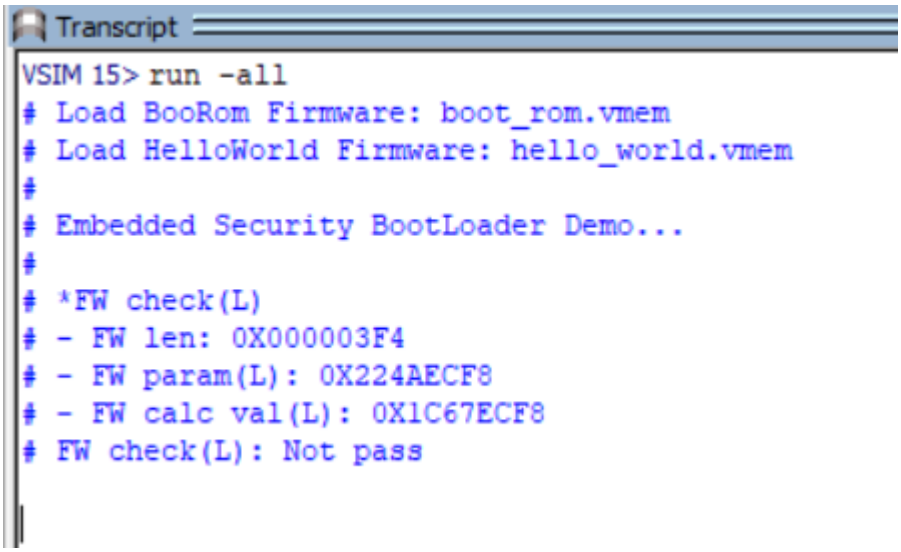
```
Transcript
VSIM 13> run -all
# Load BooRom Firmware: boot_rom.vmem
# Load HelloWorld Firmware: hello_world.vmem
#
# Embedded Security BootLoader Demo...
#
# *FW check(L)
# - FW len: 0X000003F4
# - FW param(L): 0X224AECF8
# - FW calc val(L): 0X224AECF8
# FW check(L): Pass
# *FW check(D)
# - FW param(D): 0X735F9F5F
# - FW calc val(D): 0X735F9F5F
# FW check(D): Pass
#   _    _ _____    _    _ _____    ___    _____
#  | |  | |_   _|  | |  | /  __ \  / _ \  /  ___|
#  | |_| | | | |    | |  | /  \// /_\ \ `--.
#  |  _  | | | |    | |  | |     |  _  | `--. \
#  | | | |_| |_|  | |_| | \__/\| | | |/\__/ /
#  \_| |_/\___/    \___/ \____/\_| |_/\____/
# [init] gpio ok

VSIM 14>
```

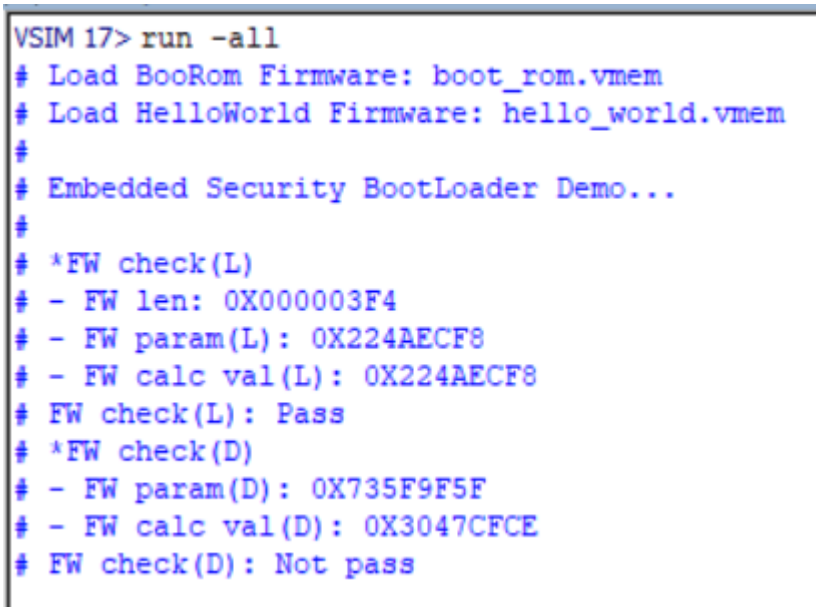
- 波形输出，符合FW中的预期



- 将FW替换为修改过的
  - 将FW中 `print_str("[init] gpio ok\r");` 修改为 `print_str("[init] gpio ok >\r");` ，FW的长度变化了，就无法通过XOR校验，也无法通过CRC32校验（在XOR无法通过后，就不进行CRC32校验了）
    - 仿真结果，无法通XOR校验



- 将FW中 `print_str("[init] gpio ok\r");` 修改为 `print_str(">init] gpio ok\r");` ，FW的长度没有变化了，可以通过XOR校验，但是无法通过CRC32校验
  - 仿真结果，无法通过CRC32校验



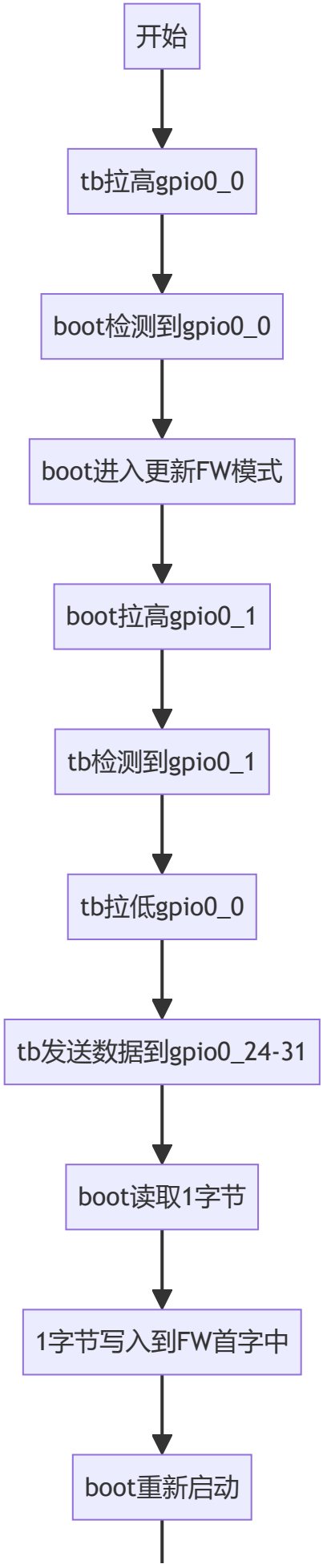
- 模拟boot中update FW存在漏洞，通过io（或例如i2c、spi等）写入了恶意FW。
  - 将 `soc_top_tb.v` 内容复制到新建的 `soc_top_tb_with_updateFW.v` 中，修改部分内容，实现如下测试

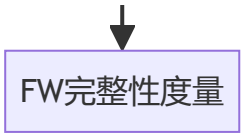
```
...
initial begin
    tb_clock   = 1'b0;
    tb_reset   = 1'b0;
    tb_uart_rx = 1'b0;
    tb_gpio_i  = 32'h00000001;

    #1000
    tb_reset   = 1'b1;
end
...
always #41.67 tb_clock = ~tb_clock;

always @(posedge tb_clock) begin
    if (tb_gpio_o[1] == 1'b1) begin
        tb_gpio_i = 32'hAA550000;
    end
end
...
```

- `soc_top_tb_with_updateFW.v` 在复位后一段时间，将gpio0 pin0拉高(置1)，然后一直检测gpio0 pin1上电平
- boot检测到gpio0 pin0高电平，将gpio0 pin1拉高，然后从gpio pin24-31 接收1字节
- `soc_top_tb_with_updateFW.v` 中检测gpio pin1拉高后，将gpioo pin0拉低，并将1字节数据输出到gpio pin24-31(这里输出的是0xAA55)
- boot接收到1字节后，将这1字节写入到FW的第一个字节中，这样就导致了FW被篡改
- boot更新FW后，跳转到boot启动地址重新启动，并进行完整性度量
- 上面的测试流程描述


开始
tb拉高gpio0_0
boot检测到gpio0_0
boot进入更新FW模式
boot拉高gpio0_1
tb检测到gpio0_1
tb拉低gpio0_0
tb发送数据到gpio0_24-31
boot读取1字节
1字节写入到FW首字中
boot重新启动



- 仿真结果
  - 串口输出

```
VSIM 21> run -all
# Load BooRom Firmware: boot_rom.vmem
# Load HelloWorld Firmware: hello_world.vmem
#
# Embedded Security BootLoader Demo...
#
# update FW ...
#
#
# update ok, reboot...
#
#
# Embedded Security BootLoader Demo...
#
# *FW check(L)
# - FW len: 0X000003F4
# - FW param(L): 0X224AECF8
# - FW calc val(L): 0XA81FECFC
# A time value could not be extracted from the current line
# FW check(L): Not pass
```

  - 波形图，输出与预期一致