

武汉大学网络安全学院

《嵌入式系统安全实验》课程

实验报告

实习题目：\_\_\_\_\_基于密码访问的门锁系统\_\_\_\_\_

专业（班）：\_\_\_\_\_16 级信安 4 班\_\_\_\_\_

学生学号：\_\_\_\_\_2016301500327\_\_\_\_\_

学生姓名：\_\_\_\_\_肖 轩 淦\_\_\_\_\_

任课教师：\_\_\_\_\_丁 玉 龙\_\_\_\_\_

2 0 1 8 年 1 2 月 1 3 日

## 目 录

第一部分	功能描述	.....	1
第二部分	设计方案	.....	
第三部分	安全设计描述	.....	
第四部分	软件设计	.....	
第五部分	设计总结	.....	

# 第一部分 功能描述

本次实验选题为基于密码访问的门锁系统，类似于现代智能家居中的智能门锁。在本次实验中，我实现了通过键盘输入密码，若正确开启门锁（电机），若错误给出提示；并通过 LED 显示提示语以及密码输入的结果；还可以通过连续敲击特殊键，输入正确密码和新密码实现密码的更改。

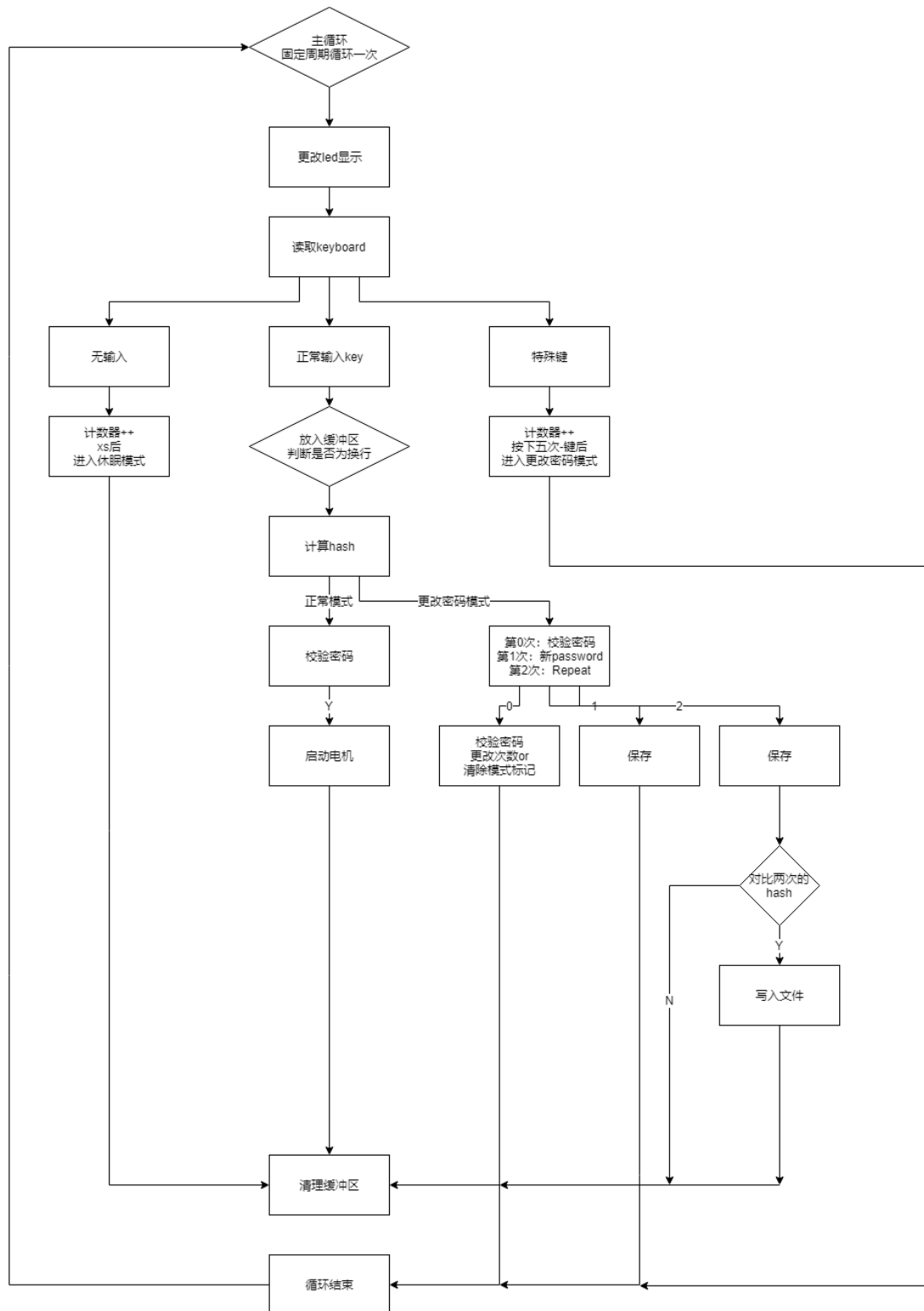
# 第二部分 设计方案

为了达到实验目标，在本次实验中我编写了两个程序：主程序 mylock 以及监控程序 lock\_guard，接下来分开介绍两个程序的设计方案。

## 一、 mylock 程序

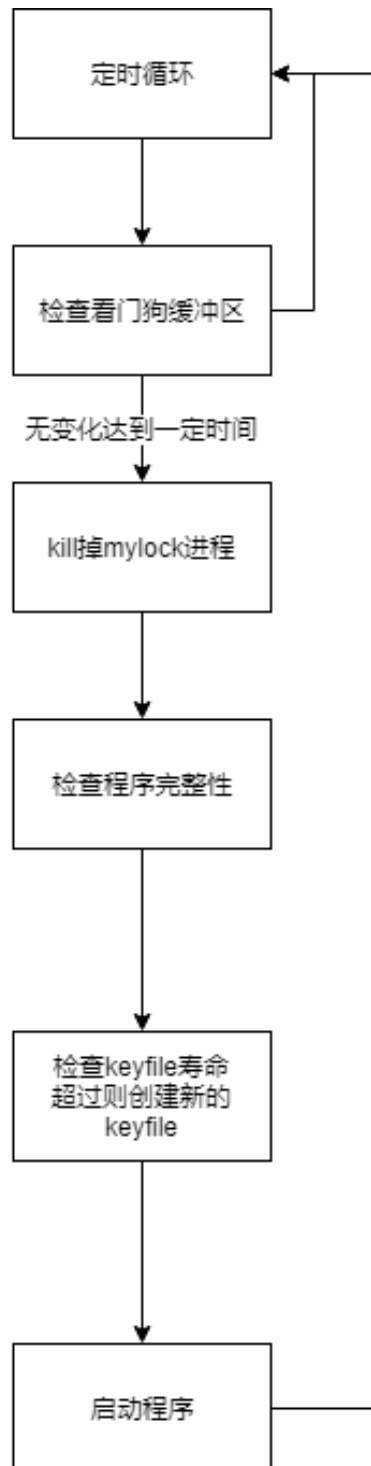
使用了双线程，其中主线程负责接受输入以及进行 hash 校验等，led 线程负责控制 led 输出相应的提示语及结果。

主线程的主要架构如下(mylock/main.c)：



## 二、 lock\_guard 监控程序

功能有检查 mylock 程序的完整性，软件看门狗，检查密钥文件使用次数。  
(lock\_guard/main.c)



## 第三部分 安全设计描述

接下来从抗干扰设计、容错设计、抗攻击设计三个方面来介绍安全设计描述：

### 一、 抗干扰设计

#### 1、复位

因为本次实验的选题为基于密码访问的门锁系统，结合现实情况考虑，采取了软件复位，没有使用人工复位。

软件复位在 lock\_guard/main.c main 函数中实现，复位分为两种：mylock 进程重启、整个系统重启。

当看门狗缓冲区达到 WCD\_RESTART\_TIME 次未发生变化时，lock\_guard 将重新启动 mylock 进程；当连续重启 mylock 进程 MAX\_TIME\_RESTART\_FAIL 次仍未成功时，lock\_guard 将记录日志并重新启动系统。

```
C:\Users\肖\Desktop\大三上\嵌入式\mylock_v2_test\lock_guard\main.c - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 格式(M) 语言(L) 设置(I) 宏(O) 运行(R) 插件(P) 窗口(W) ?

main.c
118 int main(){
119     int count_wcd = 0;
120     printf("[INFO] Lock_Guard Start!\n");
121
122     while(1){
123         if(getCurrentTime() % LOCK_GUARD_TIME_SLICE){
124
125
126             //检查缓冲区
127             if(check_wcd_buf() == 0){
128                 wcd_wait_time = 0;
129                 restart_time_sum = 0;
130                 continue;
131             }
132
133             //buf无变化
134             wcd_wait_time ++;
135             if(wcd_wait_time > WCD_RESTART_TIME){
136                 //需要重启程序
137
138                 restart_process();
139                 wcd_wait_time = 0;
140                 restart_time_sum ++;
141
142                 //重启机器
143                 if(restart_time_sum > MAX_TIME_RESTART_FAIL){
144
145                     reboot_machine();
146                 }
147             }
148         }
149     }
```

#### 2、睡眠避干扰

当检测到一定时间键盘没有按键按下时，mylock 将进入到睡眠模式，LED 也进入 sleep 模式，节能并延长 LED 的寿命。

相关代码在 mylock/main.c 中实现：

```
C:\Users\肖\Desktop\大三上嵌入式\mylock_v2_test\mylock\main_v2.0.c - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 格式(M) 语言(L) 设置(T) 宏(O) 运行(R) 插件(P) 窗口(W) ?

main.c main_v2.0.c
73
74 //非睡眠模式更改led
75 if(mode_sleep == 0)
76     led_control(LED_CONTINUE);
77 #ifdef DEBUG
78
79 //接受输入
80 ch = get_key_or_null();
81
82 #ifdef DEBUG
83
84 //空输入, 开始sleep计数
85 if(ch == 0){
86     if(count_sleep >= SLEEP_TIME_COUNT){
87         mode_sleep = 1;
88         led_control(LED_SLEEP);
89     }
90     else
91         count_sleep += 1;
92     continue;
93 }
94
95 //清空sleep计数器, 关闭睡眠模式
96 count_sleep = 0;
97 mode_sleep = 0;
98
99 //特殊键, 开始changepwd计数
100 if(ch == KEY_ENTRY_CHANGE_PWD){
101 #ifdef DEBUG
102     printf("[DEBUG]\tCount count_changepwd! Count: %d\n", count_changepwd);
103 #endif
104     if(count_changepwd >= CHANGEPWD_COUNT){
105 #ifdef STATE_CHECK_OPEN

```

### 3、指令冗余及软件陷阱捕获程序

由于 arm 指令集是定长指令集, 所有机器指令向一个 Word (4Byte) 对齐, EIP 即使跑飞, 其末两位也只能为 0。

为了减少 EIP 跑飞后的影响, 我在函数与函数之间留了一定间隔并使用 NOP+CALL error\_handel() 异常处理函数来进行填充。由于目前生成的 ELF 达到 3MB+, 故只选取了几个函数的间隔进行填充。

填充的方式是使用一个较长的无用函数在两个目标函数之间占位, 编译成可执行文件后再使用脚本进行 Patch。

接下来使用 mylock/main\_misc.c 中 checkpwd()与 markdown\_newpassword()中间的函数间隙进行举例说明。

三个函数的位置如下所示:

```
//校验密码
int checkpwd(char * hash, int * sum1, int * sum2){

//此函数作为函数间隔使用
//将被手动Patch成nop+call到错误处理函数
void nop_call_1(){

//保存新密码
int markdown_newpassword(char * hash){
```

`Nop_call_1()` 占位函数使用相同的指令进行填充(为了在编译后生成相同的机器码, 方便 Patch), 在最后加上一条 `call` 异常处理函数 (因为 `call` 使用间接寻址, 让编译器帮助我们找到异常处理函数的相对地址无疑更加方便)。

在 IDA 中查看生成未剥离符号表的可执行文件：可以看到我们的填充指令具有相同的格式，我们就可以依此把这些指令的机器码都更改为 `movs r0,r0(00 00 A0 E1)`等价于 NOP。

据此，编写出用来 **patch** 的 **python** 脚本(嵌入式-**Patch\_nop+call.py**):



```

change_hex = b'\x9F\xE5\x00\x20\xA0\xE3\x02\x10\xA0\xE1\x3F\x10\xC3\xE5'
target_hex = b'\x00\x00\xA0\xE1\x00\x00\xA0\xE1\x00\x00\xA0\xE1\x00\x00\xA0\xE1'

f = open("mylock", 'rb')

content = f.read()

f.close()

content_len = len(content)

counter = 0
mode = 0

f = open("mylock_patch", "wb")

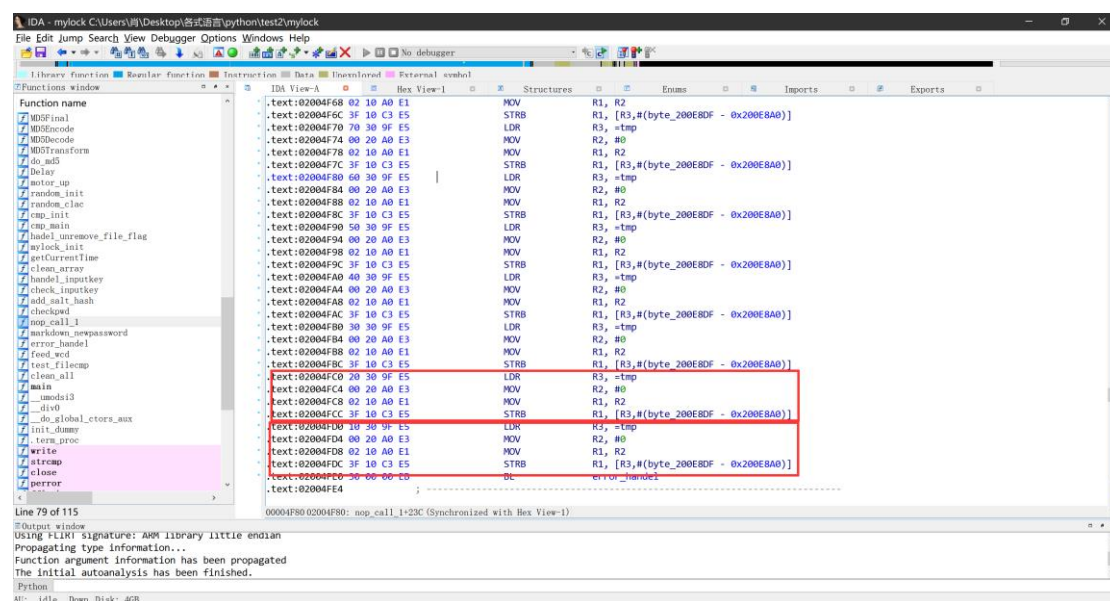
# 一条arm指令4个字节 四条arm指令16字节
for i in range(int(content_len/16)):
    if content[16*i+2: 16*(i+1)] == change_hex:
        counter += 1
        if counter > 3:
            # content[16*i: 16*(i+1)] = target_hex
            print("Patch Program %d" % i*16)
            f.write(target_hex)
        else:
            f.write(content[16*i:16*(i+1)])
    else:
        counter = 0
        f.write(content[16*i:16*(i+1)])

f.write(content[int(content_len/16)*16:])

f.close()

```

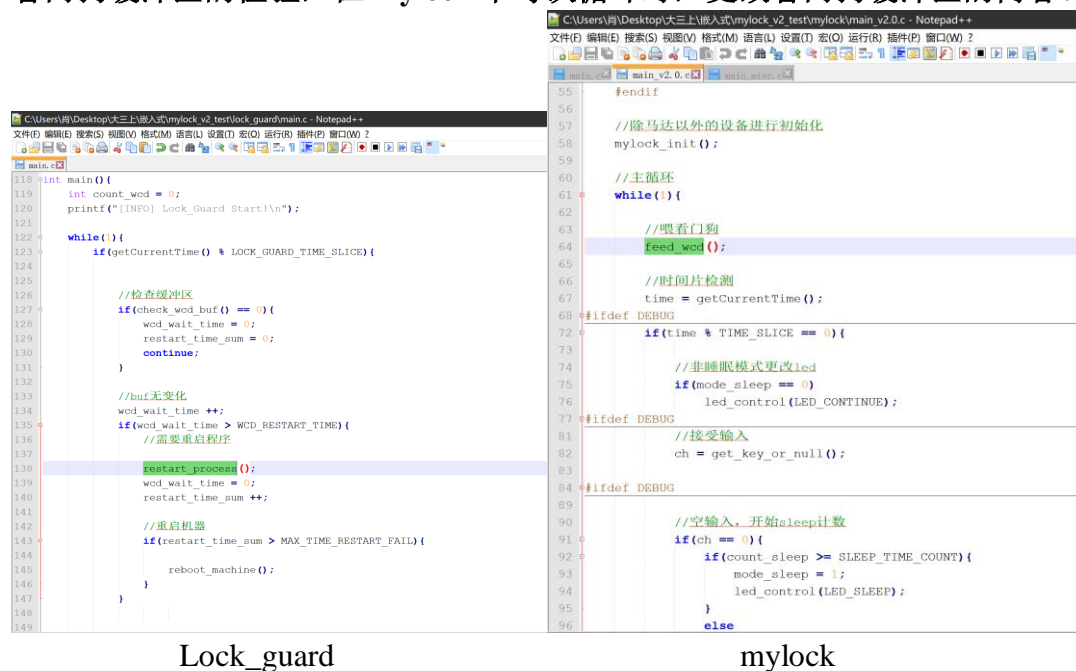
查看 patch 后的可执行文件



这样实现了函数间隙软件陷阱捕获。

#### 4、看门狗及运行监视程序

在本次实验中，lock\_guard 具有软件看门狗的功能，在 lock\_guard 中进行看门狗缓冲区的检验，在 mylock 中每次循环时，更改看门狗缓冲区的内容。



## 5、冷热启动及初始化

由于在对于密钥文件具有较为完善的修复机制使用时也有检查机制保障，使得在系统运行的任何时候断电或进程被 kill 掉，都不会有太大的影响。不过我还是将写密钥及其备份文件当作“原子操作”（只是在设置了文件标志位，但不能保证其不被打断），在写文件前设置标志，写完文件后清除标志。并在 mylock 初始化时对标志进行检查，若标志存在，则对密钥文件进行恢复。

Mylock/mylock\_init.c 中在 mylock 进行初始化时进行检查标志。

```
C:\Users\肖\Desktop\大三上\嵌入式\mylock_v2_test\mylock\mylock_init.c - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 格式(M) 语言(L) 设置(O) 运行(R) 插件(P) 窗口(W) ?

main.c main_v2.0.c main_misc.c mylock_init.c
6 int handel_unremove_file_flag(){
7     char * KEY_FILE_PATH[] = {"/keyfile/keyfile1", "/keyfile/keyfile2", "/keyfile/keyfile3"};
8     if(file_cmp(KEY_FILE_PATH, KEY_FILE_NUM) == -1){
9         printf("[ERROR]\tKEYFILE BROKEN!\n");
10        return -1;
11    }
12
13    remove_flag();
14
15    return 0;
16 }
17
18
19 int mylock_init(){
20
21     if(led_init() != 0 && kbd_init() != 0){
22         printf("[ERROR]\tInit Fail!\n");
23         exit(0);
24     }
25
26     //功率随机初始化
27     random_init();
28
29     //检查标志
30     if(check_flag() == 1)
31         if(handel_unremove_file_flag() == -1)
32             error_handel();
33
34     printf("[INFO]\tInitialization Complete!\n");
35     return 0;
36 }
```

## 6、数据备份

以磁盘上储存的 **keyfile** 及其备份文件,以及 **mylock** 可执行文件及其备份文件举例说明。

**Keyfile** 在每次读之前都会进行与其备份文件的对比并修复其中发生错误的文件,在 **mylock/file\_check/file\_cmp.c** 中实现。对比算法:程序进行对比并维护一个 **f\_num** 大小的数组,数组用来记录与其内容相同的最先一个文件的编号,当相同数量超过 **f\_num/2** 时,即认为此文件是正确的,并恢复其他不相同的文件。

**Mylock** 在每次启动前,也会由 **lock\_guard** 对 **mylock** 可执行文件进行检查,以避免其代码段或数据段发生错误而带来危害。

**Keyfile** 及其备份储存的路径为 **mylock/keyfile/**

**Mylock** 的备份位置在 **bak1/**, **bak2/**中。

因为本次实验是基于 **linux** 操作系统,**linux** 文件管理是基于 **inode**,若是自己在裸机上实现,数据备份应考虑在磁盘不同的位置进行存储,并保持较大的间隔。为了避免磁盘的一块内存区域损坏而导致全部的备份文件丢失。

## 7、重要数据包含校验码

重要数据包括:刚输入进来的明文密码、经过 **hash** 后的密码、文件密钥、电机转速等。

明文密码、**hash** 密码、文件密钥的压码规则如下,以初始密码 **123456** 为例:



```
note.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
初始密码123456

压码为(49+5+49)*3&0xFF = 53 = '5'

key+salt = "123456"+"5"+"12345678900987654321"

md5(salt) = bc88686263ef97f16c01083ff8197b35

校验位, 放在16字节hash后, 占 1 Byte, 计算式为 check_byte = sum(hash[i]) & 0xFF
压码为0x5d

XOR_KEY = 0xAA

file_content = md5^XOR_KEY = 1622c2c8c9453d5bc6aba29552b3d19f

加上压码0x5d^0xAA = 0xf7

所以最后文件内容为
1622c2c8c9453d5bc6aba29552b3d19ff7
```

电机转速校验，位于启动电机前(motor/motorup.c):

```

int motor_up(int * checkvar1, int * checkvar2)
{
    int i = 0;
    int setpwm = 100;
    int factor = DCM_TCNTB0/1024;
    int motor_time = 0;
    int count = 0;
    const int speed = setpwm * factor;
    if((dcm_fd=open(DCM_DEV, O_WRONLY))<0){
        printf("Error opening %s device\n", DCM_DEV);
        return -1;
    }

#ifdef STATE_CHECK_OPEN
    //状态检查
    check_update_state(&current_state, REQ_STATE_MOTOR_UP, STATE_MOTOR_UP);
#endif

    //再次检查对比结果
    if(*checkvar1 != *checkvar2 || *checkvar1 != 0 || *checkvar2 != 0){
        printf("unequ!\n");
        return -1;
    }

    //检查电机速度
    if((int)SPEED_CHECK != (int)speed){
        printf("speed uncorrect!\n");
        printf("speed:%d\tspeed_check:%d\n", speed, SPEED_CHECK);
        return -1;
    }

    for (; motor_time < 10; motor_time++) {

        if(count++ >= 10)
            break;

        ioctl(dcm_fd, DCM_IOCTL_SETPWM, speed);
        Delay(500);
        printf("setpwm = %d \n", setpwm);

    }

    close(dcm_fd);
    return 0;
}

```

## 二、容错设计

### 1、自检

检查程序源代码完整性的工作是在 lock\_guard 中实现(lock\_guard/main.c), 在每次启动 mylock 前都会对 mylock 的 elf 文件进行完整性检验

```

int restart_process(){

    printf("[WARNING]\tRestarting process\n");
    system("killall mylock");

    file_cmp(MYLOCK_AND_BAK_PATH, MYLOCK_FILE_CHECK_NUM);

    check_keyfile_counter();

    system("mylock/mylock");
    return 0;
}

```

## 2、参数输入容错设计

在输入时对缓冲区进行了限制，可以指定最长的密码长度，不会出现缓冲区溢出，(mylock/main.c)

```
//数字，加入输入数组，并添加压码
if('0'<=ch && ch<='9'){
    input_key[count_inputkey%KEY_LEN] = ch;
    count_inputkey = (count_inputkey+1)%KEY_LEN;
    check_inputkey_num = (check_inputkey_num + ch) & 0xFF;
    input_key[(count_inputkey%KEY_LEN)] = check_inputkey_num;

#ifdef DEBUG
#endif

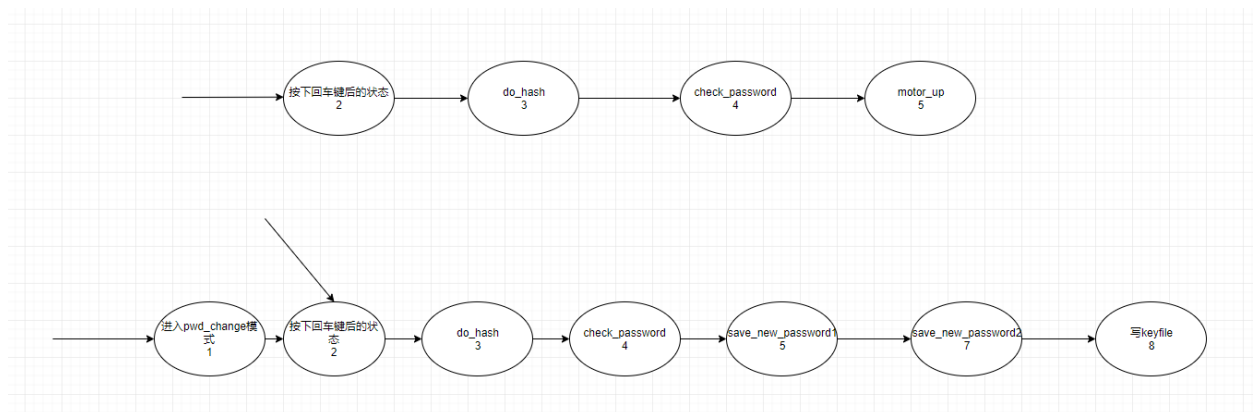
    led_control(LED_ADD_ONE_CHAR);

    continue;
}
```

## 3、输入界面的安全性设计

## 4、状态转移的条件审核

设计的状态转换图如下所示：在进入相关状态时都会进行状态的检查：



如在进入 **motor\_up** 时：

```

/*****
int motor_up(int * checkvar1, int * checkvar2)
{
    int i = 0;
    int setpwm = 100;
    int factor = DCM_TCNTB0/1024;
    int motor_time = 0;
    int count = 0;
    const int speed = setpwm * factor;
    if((dcm_fd=open(DCM_DEV, O_WRONLY))<0){
        printf("Error opening %s device\n", DCM_DEV);
        return -1;
    }

#ifdef STATE_CHECK_OPEN
    //状态检查
    check_update_state(&current_state, REQ_STATE_MOTOR_UP, STATE_MOTOR_UP);
#endif

    //再次检查对比结果
    if(*checkvar1 != *checkvar2 || *checkvar1 != 0 || *checkvar2 != 0){
        printf("unequ!\n");
        return -1;}

    //检查电机速度
    if((int)SPEED_CHECK != (int)speed){
        printf("speed uncorrect!\n");
        printf("speed:%d\tspeed_check:%d\n", speed, SPEED_CHECK);
        return -1;}

    for (; motor_time < 10; motor_time++) {

        if(count++ >= 10)
            break;

        ioctl(dcm_fd, DCM_IOCTL_SETPWM, speed);
        Delay(500);
        printf("setpwm = %d \n", setpwm);

    }

    close(dcm_fd);
    return 0;
}

```

在验收时经老师提醒，应使用状态转移序列而非使用全局整型变量。  
 在 hash、检查密钥、启动电机时添加状态转移序列的检验。  
 如在 motor 中，对之前的状态序列进行逐一检查：

```

23  /*****
24  int motor_up(int * checkvar1, int * checkvar2, int * state)
25  {
26      int i = 0;
27      int setpwm = 100;
28      int factor = DCM_TCNTB0/1024;
29      int motor_time = 0;
30      int count = 0;
31      const int speed = setpwm * factor;
32      if((dcm_fd=open(DCM_DEV, O_WRONLY))<0){
33          printf("Error opening %s device\n", DCM_DEV);
34          return -1;
35      }
36
37      #ifdef STATE_CHECK_OPEN
38          //状态检查
39          check_update_state(&current_state, REQ_STATE_MOTOR_UP, STATE_MOTOR_UP);
40          if(state[0] == 2 && state[1] == 1 && state[2] == 2){
41              state[3] = 3;
42              state[0] = 3;
43          }
44          else{
45              return -1;
46          }
47      #endif
48

```

## 5、安全存储

对密钥文件进行安全存储，明文密码经过加盐 hash 后得到真正的密钥，在存入文件时，加密后再存入文件，使用时从文件中解密再开始进行对比。

hash 密钥在内存与文件中的加解密实现得比较简单，仅使用异或进行加解密，若在真实环境中需使用分组密码及每机不同的分组密码密钥进行加密。

实现(file\_encry/keyfilemanager.c):



```

//加解密从文件中读取到的数值
void decryKeyFileStr(unsigned char * keyfiledata, unsigned char * key){
    int i = 0;
    for(; i < READ_SIZE; i++){
        key[i] = keyfiledata[i] ^ XOR_KEY;
    }
    return;
}

//读取文件并解密
int readKeyFile(char * f_path, unsigned char * key){
    int i = 0;

    //读取到未解密的key_data
    unsigned char keyfiledata[READ_SIZE];
    //读取文件是否成功的标志变量
    int readfileflag = 0;

    //打开文件，常规操作
    int fd;

    printf("enter readKeyFile!\n");

    fd = open(f_path, O_RDONLY);
    if(fd < 0)
    {
        printf("keyfilemanager.c\treadKeyFile()\t*****KEY FILE OPEN FAIL!*****\n");
        return -1;
    }

    //读取文件，并检测是否出错
    readfileflag = read(fd, keyfiledata, READ_SIZE);
    if(readfileflag <= 0){
        printf("keyfilemanager.c\treadKeyFile()\t*****KEY FILE READ ERROR!*****\n");
        return -1;
    }
    close(fd);

#ifdef DEBUG
    printf("keyfilemanager.c\treadKeyFile()\t");
    for(i=0;i<READ_SIZE;i++)
    {
        printf("%02x",keyfiledata[i]);
    }
    printf("\n");
#endif

    //解密读取到的字符串，并copy至传入的地址中，准备传出
    decryKeyFileStr(keyfiledata, key);

    //清除局部变量
    for(i = 0; i < READ_SIZE; i++)
        keyfiledata[i] = 0;
    fd = 0;

    return 0;
}

```

### 三、 抗攻击设计

对于旁路攻击的防御：

#### 1、 数据冗余

文件密钥的数据单元及压码在之前均有介绍，储存位置随机暂未实现。

#### 2、 控制冗余

重要函数入口序列检查在之前有过介绍

#### 3、 执行冗余

在程序中，容易受到旁路攻击的是 hash 过程，及密码 hash 检查的过程。我在这两处加入了随机数据计算进行干扰。

(1) hash 部分，由于采用的是 md5 散列算法，md5 中包含大量的移位异或，故在每轮变换时插入随机次异或来进行混淆。

```
void random_md5(){
    int i,j,k;

    j = 0;
    k = 0;

    for(i = 0; i < random(10); i++){
        j = random(0xFFFFFFFF) ^ random(0xFFFFFFFF);

    }
    return;
}
```

```
#define F(x,y,z) ((x & y) | (~x & z))
#define G(x,y,z) ((x & z) | (y & ~z))
#define H(x,y,z) (x*y^z)
#define I(x,y,z) (y ^ (x | ~z))
#define ROTATE_LEFT(x,n) ((x << n) | (x >> (32-n)))
#define FF(a,b,c,d,x,s,ac) { a += F(b,c,d) + x + ac; random_md5(); a = ROTATE_LEFT(a,s); a += b; }
#define GG(a,b,c,d,x,s,ac) { a += G(b,c,d) + x + ac; random_md5(); a = ROTATE_LEFT(a,s); a += b; }
#define HH(a,b,c,d,x,s,ac) { a += H(b,c,d) + x + ac; random_md5(); a = ROTATE_LEFT(a,s); a += b; }
#define II(a,b,c,d,x,s,ac) { a += I(b,c,d) + x + ac; random_md5(); a = ROTATE_LEFT(a,s); a += b; }

void MD5Init(MD5_CTX *context);
void MD5Update(MD5_CTX *context,unsigned char *input,unsigned int inputlen);
void MD5Final(MD5_CTX *context,unsigned char digest[16]);
void MD5Transform(unsigned int state[4],unsigned char block[64]);
void MD5Encode(unsigned char *output,unsigned int *input,unsigned int len);
void MD5Decode(unsigned int *output,unsigned char *input,unsigned int len);
```

```
pthread.c  x  main_v2.0.c  x  md5.c  x  do_md5.c
88 FF(d, a, b, c, x[ 1], 12, 0xe8c7b756); /* 2 */
89 FF(c, d, a, b, x[ 2], 17, 0x242070db); /* 3 */
90 FF(b, c, d, a, x[ 3], 22, 0xc1bdceee); /* 4 */
91 FF(a, b, c, d, x[ 4], 7, 0xf57c0faf); /* 5 */
92 FF(d, a, b, c, x[ 5], 12, 0x4787c62a); /* 6 */
93 FF(c, d, a, b, x[ 6], 17, 0xa8304613); /* 7 */
94 FF(b, c, d, a, x[ 7], 22, 0xfd469501); /* 8 */
95 FF(a, b, c, d, x[ 8], 7, 0x690090d8); /* 9 */
96 FF(d, a, b, c, x[ 9], 12, 0x8b44f7af); /* 10 */
97 FF(c, d, a, b, x[10], 17, 0xffff5bb1); /* 11 */
98 FF(b, c, d, a, x[11], 22, 0x895cd7be); /* 12 */
99 FF(a, b, c, d, x[12], 7, 0x6b901122); /* 13 */
100 FF(d, a, b, c, x[13], 12, 0xfd987193); /* 14 */
101 FF(c, d, a, b, x[14], 17, 0xa679438e); /* 15 */
102 FF(b, c, d, a, x[15], 22, 0x49b40821); /* 16 */
103
104 /* Round 2 */
105 GG(a, b, c, d, x[ 1], 5, 0xf61e2562); /* 17 */
106 GG(d, a, b, c, x[ 6], 9, 0xc040b340); /* 18 */
107 GG(c, d, a, b, x[11], 14, 0x265e5a51); /* 19 */
108 GG(b, c, d, a, x[ 0], 20, 0xe9b6c7aa); /* 20 */
109 GG(a, b, c, d, x[ 5], 5, 0xd62f105d); /* 21 */
110 GG(d, a, b, c, x[10], 9, 0x2441453); /* 22 */
111 GG(c, d, a, b, x[15], 14, 0xd8a1e681); /* 23 */
112 GG(b, c, d, a, x[ 4], 20, 0xe7d3fbc8); /* 24 */
113 GG(a, b, c, d, x[ 9], 5, 0x21e1cde6); /* 25 */
114 GG(d, a, b, c, x[14], 9, 0xc33707d6); /* 26 */
115 GG(c, d, a, b, x[ 3], 14, 0xf4d50db7); /* 27 */
116 GG(b, c, d, a, x[ 8], 20, 0x455a14ed); /* 28 */
117 GG(a, b, c, d, x[13], 5, 0xa9e3e905); /* 29 */
118 GG(d, a, b, c, x[ 2], 9, 0xfcefa3f8); /* 30 */
119 GG(c, d, a, b, x[ 7], 14, 0x676f02d9); /* 31 */
120 GG(b, c, d, a, x[12], 20, 0x8d2a4c8a); /* 32 */
121
122 /* Round 3 */
123 HH(a, b, c, d, x[ 5], 4, 0xffffa3942); /* 33 */
124 HH(d, a, b, c, x[ 8], 11, 0x8771f681); /* 34 */
125 HH(c, d, a, b, x[11], 16, 0x6d9d6122); /* 35 */
126 HH(b, c, d, a, x[14], 23, 0xfde5380c); /* 36 */
127 HH(a, b, c, d, x[ 1], 4, 0xa4beea44); /* 37 */
128 HH(d, a, b, c, x[ 4], 11, 0x4bdecfa9); /* 38 */
129 HH(c, d, a, b, x[ 7], 16, 0xf6bb4b60); /* 39 */
130 HH(b, c, d, a, x[10], 23, 0x28ebfbc70); /* 40 */
131 HH(a, b, c, d, x[13], 4, 0x289b7ec6); /* 41 */
132 HH(d, a, b, c, x[ 0], 11, 0xeaa127fa); /* 42 */
133 HH(c, d, a, b, x[ 3], 16, 0xd4ef3085); /* 43 */
134 HH(b, c, d, a, x[ 6], 23, 0x4881d05); /* 44 */
135 HH(a, b, c, d, x[ 9], 4, 0xd9d4d039); /* 45 */
136 HH(d, a, b, c, x[12], 11, 0xe6db99e5); /* 46 */
137 HH(c, d, a, b, x[15], 16, 0x1fa27cf8); /* 47 */
138 HH(b, c, d, a, x[ 2], 23, 0xc4ac5665); /* 48 */
139
140 /* Round 4 */
141 II(a, b, c, d, x[ 0], 6, 0xf4292244); /* 49 */
142 II(d, a, b, c, x[ 7], 10, 0x432aff97); /* 50 */
143 II(c, d, a, b, x[14], 15, 0xab94237a); /* 51 */
144 II(b, c, d, a, x[ 5], 21, 0xfc93a039); /* 52 */
145 II(a, b, c, d, x[12], 6, 0x655b59c3); /* 53 */
146 II(d, a, b, c, x[ 3], 10, 0x8f0ccc92); /* 54 */
147 II(c, d, a, b, x[10], 15, 0xffefff47d); /* 55 */
148 II(b, c, d, a, x[ 1], 21, 0x85845dd1); /* 56 */
149 II(a, b, c, d, x[ 8], 6, 0x6fa87e4f); /* 57 */
```

## (2) 密钥对比过程

对比时使用的是加以及异或，故也使用加随机数、异或随机数进行混淆。

```
14 void random_clac(){
15     int i, j, k;
16
17     j = 0;
18     k = 0;
19
20     for(i = 0; i < random(100)+20; i++)
21         j += random(1024) ^ random(1024);
22
23     return;
24 }
```

```
168 //采用异或对hash
169 while(i < MAX_FILE_SIZE){
170     if(hash[i] == 0 || file_key[i] == 0)
171         break;
172     *sum1 += hash[i] ^ file_key[i];
173     *sum2 += hash[i] ^ file_key[i];
174
175     //随机计算
176     random_clac();
177
178     i++;
179 }
```

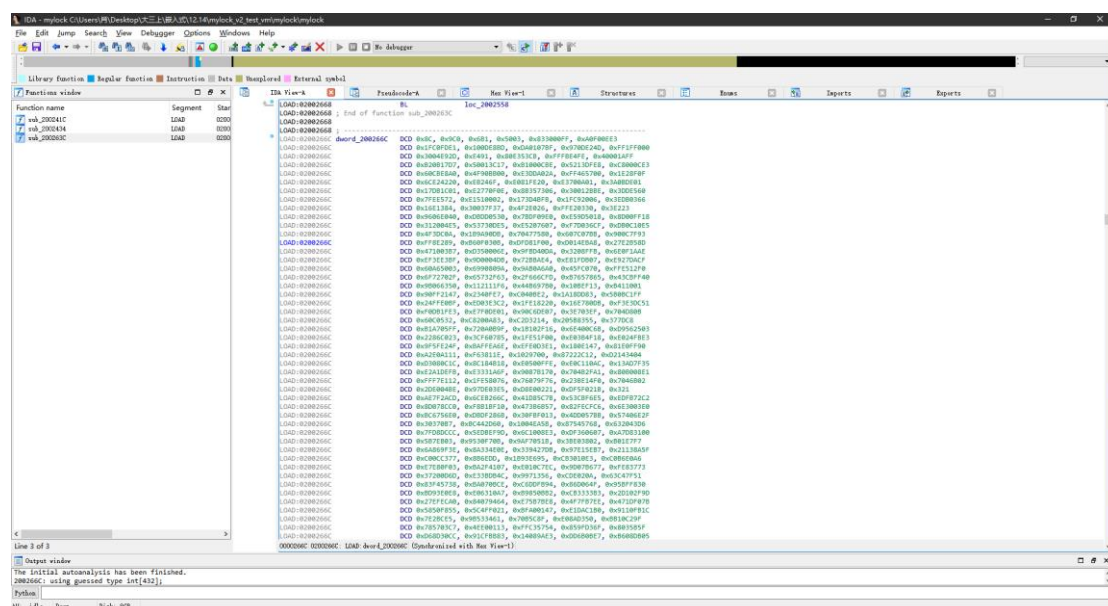
## 4、掩码技术（加盐）

对于输入的较短的明文密码进行加盐 hash，避免攻击者获取到 hash 值后进行彩虹表攻击。

对于逆向工程的防御：

### 1、软件加壳

通过 upx 加壳，减小了程序空间，且给攻击者在得到 elf 文件后的逆向工程增加难度。



对于爆破密码的防御:

### 1、设置密码错误次数上限

在进行密码 hash 检查时，会记录失败的次数，若达到上限，则拒绝检查密码，不会解密密钥文件。该次数在一定时间后清零。

```
128 //校验密码
129 int checkpwd(char * hash, int * sum1, int * sum2){
130     int i = 0, debug;
131     char file_key[MAX_FILE_SIZE];
132     memset(file_key, 0, sizeof(file_key));
133
134 #ifdef STATE_CHECK_OPEN
135     //状态检查
136     check_update_state(&current_state, REQ_STATE_CHECK_PSW_ED, STATE_CHECK_PSW_ED);
137 #endif
138
139     //使其汉明码距最大，发生错误造成的损失最小
140     *sum1 = -1;
141     *sum2 = -1;
142
143     //是否已达到比较密码失败的上限
144     if(current_check_psw_fail >= MAX_CHECK_PSW_FAIL)
145         return 0;
146
147
148     //对密码文件进行对比
149     if(file_cmp(KEY_FILE_PATH, KEY_FILE_NUM) == -1){
150         printf("[ERROR]\tKEYFILE BROKEN!\n");
151         return -1;
152     }
153
154 #ifdef DEBUG
155     printf("[INFO]\tReading KeyFile!\n");
156 #endif
157
158     //读密码文件
159     readKeyFile(KEY_FILE_PATH[0], file_key);
160
161 #ifdef DEBUG
162     printf("[INFO]\tRead KeyFile Succ!");
163     for(debug = 0; debug < HASH_LEN+5; debug++){
164         printf("%2x", file_key[debug]);
165     }
166     printf("\n");
167 #endif
```

对于硬件攻击的防御：（注：这里硬件攻击指的是攻击者可以在系统运行的某个时刻获取系统的内存或磁盘切片）

### 1、密钥文件再加密

在前面已经提到密钥在内存中进行对比和在文件中存储是会经过加解密。

### 2、敏感变量及时清理

敏感变量指的是在储存输入的明文密码、压码、hash 值等中间变量。在完成其功能后立即对其进行清理（填充 0），避免攻击者在合法用户输入正常密码离开后得到内存切片直接获取相关信息。

在代码中较多地方都有体现，以处理明文密码缓冲区为例(mylock/main.c):

在进行 hash 后立即用 0 覆盖明文密码缓冲区及其压码。

```

148         //回车键, next
149         if(ch == 13){
150 #ifdef STATE_CHECK_OPEN
151             //状态检查
152             check_update_state(&current_state, REQ_STATE_ENTER_ED, STATE_ENTER_ED);
153 #endif
154             //将末尾用0填充
155             handel_inputkey(input_key, count_inputkey + 1);
156             check_inputkey_num = 0;
157
158 #ifdef DEBUG ***
159 #endif
160
161             //出现问题
162             if(add_salt_hash(input_key, hash) == -1){
163 #ifdef DEBUG
164                 printf("[INFO]\tADD_SALT_HASH ERROR!\n");
165 #endif
166                 clean_all();
167                 led_control(LED_PSW_WRONG);
168                 continue;
169             }
170
171 #ifdef DEBUG ***
172 #endif
173
174             //清理明文信息
175             clean_array(input_key);
176             count_inputkey = 0;
177
178
179

```

## 第四部分 软件设计

操作流程（本机截图仅为暂时目录，实际应在试验台上进行操作）：

1、在 mylock 文件夹及 lock\_guard 文件夹中 make 成功后，进入主文件夹：

此电脑 > 桌面 > 大三上 > 嵌入式 > mylock

名称	修改日期	类型	大小
bak1	2018/12/14 17:08	文件夹	
bak2	2018/12/14 17:08	文件夹	
lock_guard	2018/12/17 22:07	文件夹	
mylock	2018/12/17 22:07	文件夹	
wcd_buf	2018/12/17 22:07	文件夹	
init.sh	2018/12/15 11:58	Shell Script	1 KB
install.sh	2018/12/18 0:14	Shell Script	1 KB
readme.txt	2018/12/15 11:45	文本文档	1 KB

2、执行 install.sh 安装相关驱动

3、执行 init.sh 将 mylock 文件 copy 覆盖其备份（否则启动 lock\_guard 会将新生成的 elf 文件误认为是发生错误的 elf）

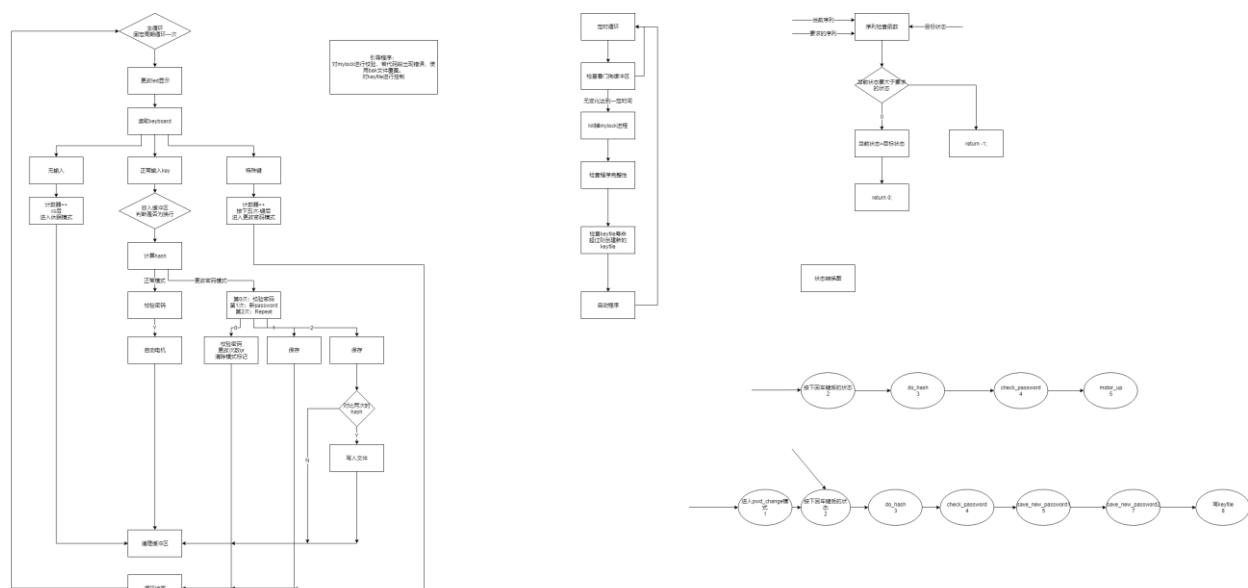
4、在主文件夹下执行 ./lock\_guard/lock\_guard 启动监控进程，监控进程会启动 mylock，或使用脚本 ./run.sh

- 5、试验台上 led 会出现相关提示语，默认密码 123456 回车即可启动电机
- 6、连续按下 6 次小键盘上的 '.' 键进入更换密码模式
- 7、输入原密码 123456
- 8、输入新密码 如 000000
- 9、重复输入新密码 000000，若重复时输错将回到第 8 步
- 10、使用新密码 000000 启动电机
- 11、CTRL+C 退出 mylock 进程，lock\_guard 会自动重启 mylock，此时新密码 000000 依然有效。

（文件采用相对路径，所以启动进程操作须在主目录下完成）

功能架构图：

包括主进程、监控进程、状态转换图、序列校验算法图



密钥加密流程(mylock/note.txt):

```
note.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
初始密码123456

压码为(49+5+49)*3&0xFF = 53 = '5'

key+salt = "123456"+"5"+"12345678900987654321"

md5(salt) = bc88686263ef97f16c01083ff8197b35

校验位, 放在16字节hash后, 占 1 Byte, 计算式为 check_byte = sum(hash[i]) & 0xFF
压码为0x5d

XOR_KEY = 0xAA











file_content = md5^XOR_KEY = 1622c2c8c9453d5bc6aba29552b3d19f

加上压码0x5d^0xAA = 0xf7

所以最后文件内容为
1622c2c8c9453d5bc6aba29552b3d19ff7
```

附件文件说明

主文件夹:

	bak1	2018/12/14 17:08	文件夹	
	bak2	2018/12/14 17:08	文件夹	
	lock_guard	2018/12/18 0:29	文件夹	
	mylock	2018/12/18 0:31	文件夹	
	other	2018/12/18 0:33	文件夹	
	wcd_buf	2018/12/17 22:07	文件夹	
	init.sh	2018/12/15 11:58	Shell Script	1 KB
	install.sh	2018/12/18 0:14	Shell Script	1 KB
	readme.txt	2018/12/18 0:27	文本文档	1 KB
	run.sh	2018/12/18 0:26	Shell Script	1 KB

Bak1: mylock 二进制文件的备份 1

Bak2: mylock 二进制文件的备份 2

Lock\_guard: 监控进程






















Mylock: 主进程

Other: 其中用到的一些 python 脚本, 包括密钥文件的计算生成等等

Wcd\_buf: 看门狗缓冲区

(Upx 加壳工具因较大且网上有就没放进来了)

Mylock 文件夹

 drivers	2018/12/17 22:07	文件夹	
 file_check	2018/12/18 0:29	文件夹	
 file_encry	2018/12/18 0:30	文件夹	
 keyboard	2018/12/18 0:30	文件夹	
 keyfile	2018/12/17 22:07	文件夹	
 led	2018/12/18 0:30	文件夹	
 md5	2018/12/18 0:30	文件夹	
 motor	2018/12/18 0:30	文件夹	
 state_check	2018/12/18 0:30	文件夹	
 test	2018/12/17 22:07	文件夹	
 cmp_hash.c	2018/12/15 11:47	C Source file	2 KB
 global_var.c	2018/12/12 17:42	C Source file	1 KB
 led_thread.c	2018/12/17 13:34	C Source file	3 KB
 main_misc.c	2018/12/18 0:00	C Source file	6 KB
 main_v2.0.c	2018/12/18 0:00	C Source file	8 KB
 Makefile	2018/12/17 11:00	文件	1 KB
 mylock.log	2018/12/14 16:06	文本文档	1 KB
 mylock_config.h	2018/12/17 10:52	C++ Header file	2 KB
 mylock_init.c	2018/12/17 10:58	C Source file	1 KB
 note.txt	2018/12/17 0:04	文本文档	1 KB
 random_clac.c	2018/12/16 23:17	C Source file	1 KB

## 第五部分 设计总结

通过本次实验，通过自己设计一个嵌入式系统并对其进行安全加固，让我对于嵌入式系统安全，对于其抗干扰设计、容错设计、抗攻击设计有了更深刻的了解。