

学号 2015302580343

密级 公 开

武汉大学本科毕业论文

个人文件加密系统

院（系）名 称：计算机学院

专 业 名 称 ： 软件工程

学 生 姓 名 ： 张涛

指 导 教 师 ： 徐武平 副教授

二〇一九年四月

**BACHELOR'S DEGREE THESIS
OF WUHAN UNIVERSITY**

Personal File Encryption System

College : Computer School

Major : Software Engineering

Name : Tao Zhang

Supervisor : Dr. Wuping Xu

April 2019

郑 重 声 明

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名：_____ 日期：_____

摘 要

本文实现了一个个人文件加密系统，提供多级密钥管理，动态修改可执行文件以获得每个用户使用不同的主密钥;通过各种关键信息计算出用户密钥而非直接记录用户密钥的加密文本，拥有较难破解的用户密钥；使用自定义个文件密钥加密文件；而且提供不需要解密而直接通过网络进行访问的方式，当然，本身不包含任何访问公共网络的逻辑，以保证用户放心。

关键字：个人加密系统 openssl 网络访问 修改动态链接库

ABSTRACT

I have implemented a personal file encryption system, which provides multi-level key management. It will dynamically modify the executable file to make different have different master keys. It will calculate the user key through various information instead of directly recording the encrypted text on the disk, which makes the user hardly to crack. Using a custom file key to encrypt the file makes it more convenient. It also provides functions to access encrypted files through the network without decryption. Of course, it does not contain any access to the public network logic, because it may make the user worry about the reliability;

Keywords: personal file encryption system, openssl, access encrypted files through the network without decryption, modify executable file

第一章 背景与意义	1
1.1 研究的原因	1
1.2 国内的研究现状和发展趋势	1
1.2.1 国内的文件加密软件	1
1.2.2 国内常见的加密软件	3
第二章 加密算法对比与选择	4
2.1 对称加密与非对称加密	4
2.2 各种对称加密算法的比较	5
2.2.1 DES 加密算法	5
2.2.2 3DES 加密算法	5
2.2.2 IDEA 加密算法	5
2.2.2 AES 加密算法	5
2.2.2 各种对称加密算法的比较	6
2.3 分组密码工作模式	6
2.3.1 ECB 电子密码本	6
2.3.2 CBC 密文链接	6
2.3.3 其他工作模式与总结	6
2.3 算法库	7
第三章 工作环境搭建	7
3.1 总体说明	7
3.2 依赖及编译/获取方式:	8
3.3 自动加载子模块的 cmake 脚本	8
3.4 版本控制	10
第四章 测试驱动	10
第五章 系统总体设计	13
5.1 总体架构	13
5.2 总体模块说明	14
5.2.1 项目主目录	14
5.2.2 common: 包含通用的工具, 配置文件, 日志适配器等	14
5.2.3 src: 各个功能模块动态链接库和测试可执行文件的实现	14
5.2.4 gui: 通过命令模式驱动的 qt gui 的实现	14
5.3 生成的可执行文件说明	15
第六章 各个模块的详细说明与实现方式	16
6.1. common/common_include 通用方法和头文件	16
6.2 src/crypto_adaptor: 加密解密算法适配器	17
6.3 src/file_coltroller: 文件管理器的逻辑实现	21
6.4 修改动态链接库的主密钥:	26

6.5 src/passwd_adaptor: 用户密钥与文件密钥的管理器.....	30
6.6 src/gui_qt 利用 QT 库实现一个图形界面.....	33
6.6.1 gui 实现方式对比与分析.....	33
6.6.2 命令模式主控的实现.....	35
6.6.3 命令行客户端的实现.....	38
6.6.4 qt 客户端的实现.....	39
6.7 src/web_adaptor: 简单的网络实时解密浏览实现.....	43
第七章 不足与期待.....	45
7.1 实现一个 NFS 服务器.....	45
7.2 脱离 cygwin 而实现全平台编译.....	45
7.3 对网路访问进行权限验证.....	46
7.3.1 自己实现简单的网络链接管理.....	47
7.3.2 使用 apache 服务器的 cgi 功能.....	47
7.4 网络穿透.....	47
7.4.1 NET 转换.....	47
7.4.2 反向代理.....	48
7.5 文件压缩.....	48
6.1 LZMA SDK includes:.....	48
20-50 MB/s on modern 3 GHz CPU (Intel, AMD, ARM).....	49
第八章 结语.....	49
参考文献.....	50
致谢.....	51

第一章 背景与意义

1.1 研究的原因

个人文件加密系，在办公区等工种场所，或者比较重要的个人文件（如密码本）等方面都有需求。移动端一般厂商自带加密功能，且被攻破的机会比较少（在没有 root 的情况下，厂商自带的加密系统文件是无法访问到的）；macOS/linux 有比较完善的用户权限系统，并不太需要。

但是 windows 端则完全相反。

(1) windows 的文件权限系统设计并不友好，普通用户几乎没什么人用，windows 自带的加密时针对磁盘的，而没有类似 lvm 的技术支持，这会非常笨拙，危险；

(2) 在完全单机的情况下，如何保护主密钥是一个很困难的问题，应为它就存储在本机，总能被找到，破解；

(3) java/c#/python 的加密/混淆机制并没有太好的作用，几乎是可以看到源码的，c++要好一些，但是也有反汇编然后修改关键函数返回的破解存在；

(4) 加密过的文件在使用时得经过解密，写回磁盘才可以使用，但对大文件而言这一过程往往比较耗时，尤其是只读取文件时，写到磁盘显得毫无作用。

由于以上原因，市面上目前没有太好的相关软件，要么收费且不方便，要么必须联网，难以信任，难以推广。因此设计一款比较难破解，非常方便使用的个人文件加密系统就很有必要。

1.2 国内的研究现状和发展趋势

1.2.1 国内的文件加密软件

护密文件夹加密软件大师，主界面如图 1.1：

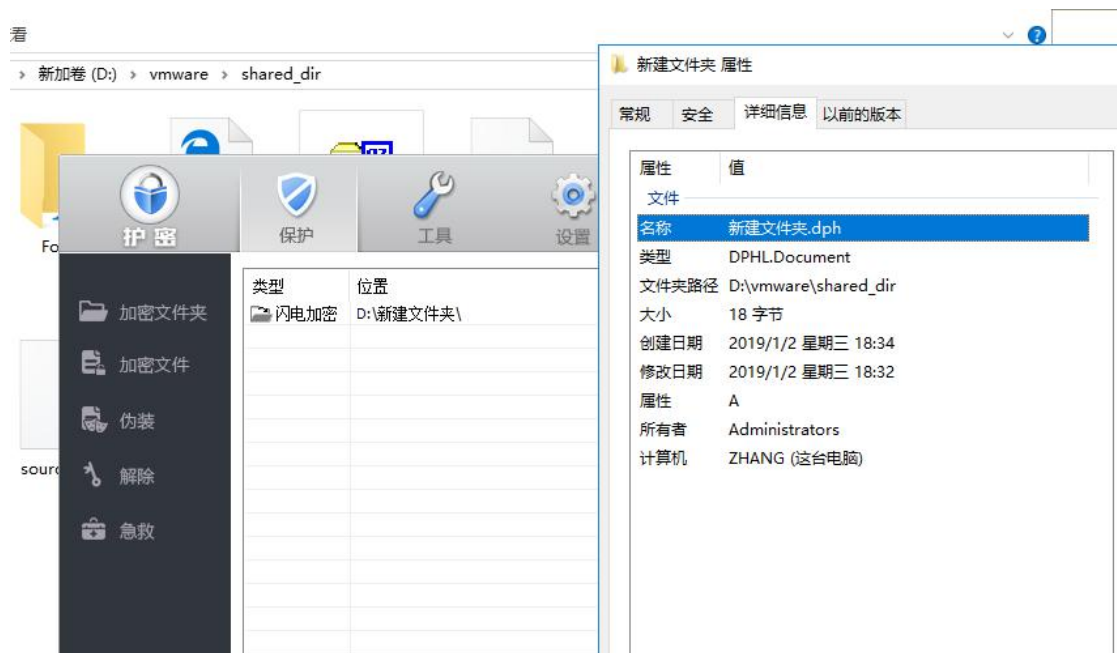


图 1.1 护密文件夹加密软件大师主界面和创建的. dph 文件

优点：小巧（4M）不依赖网络，在原位置创建. dph 文件以链接到程序；缺点：单级密码，安全度不高，文件建相互依赖时不好处理。

红线隐私保护系统控制界面如图 1.2：

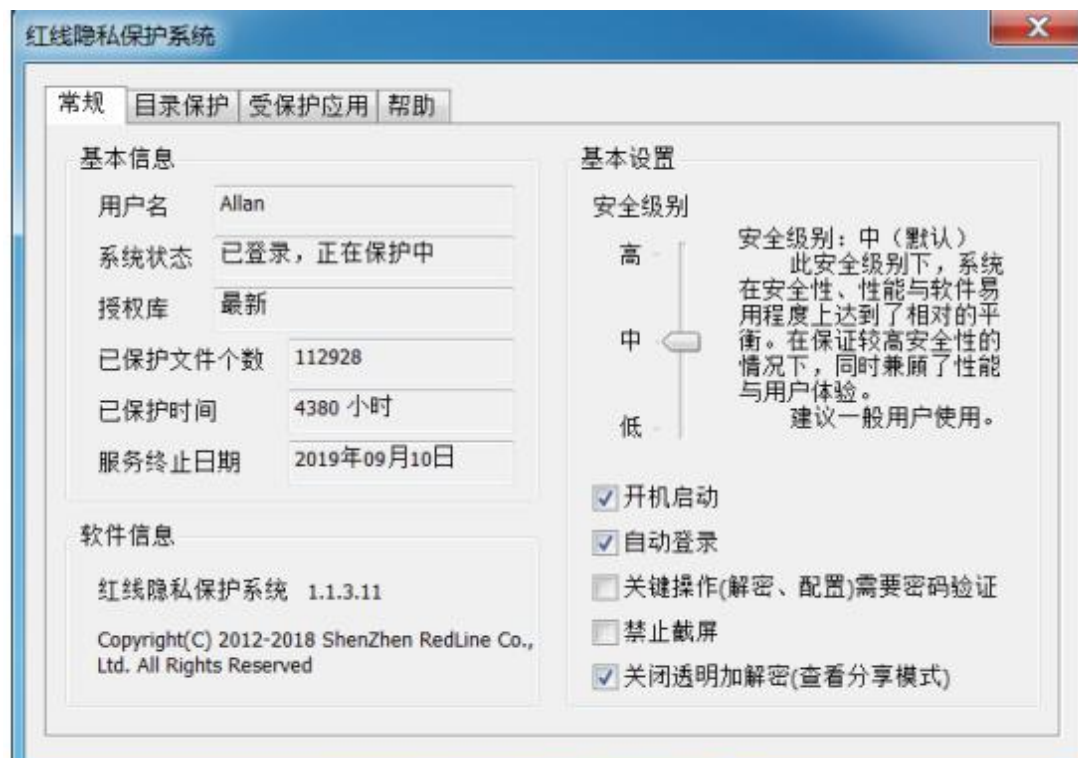


图 1.2 红线隐私保护系统

优点：可以直接打开被加密的文档进行编辑修改操作。无需手动解密再编辑，

无需编辑保存后再次加密。使用方便，加密复杂，安全度高；缺点：是一款企业级软件的个人移植版，需要购买，需要联网，是一款半在线的加密软件，你的数据会被同步到它的服务器上，主密钥也会存在服务器上。

1.2.2 国内常见的加密软件

VeraCrypt，挂载虚拟磁盘设置如图 1.3:

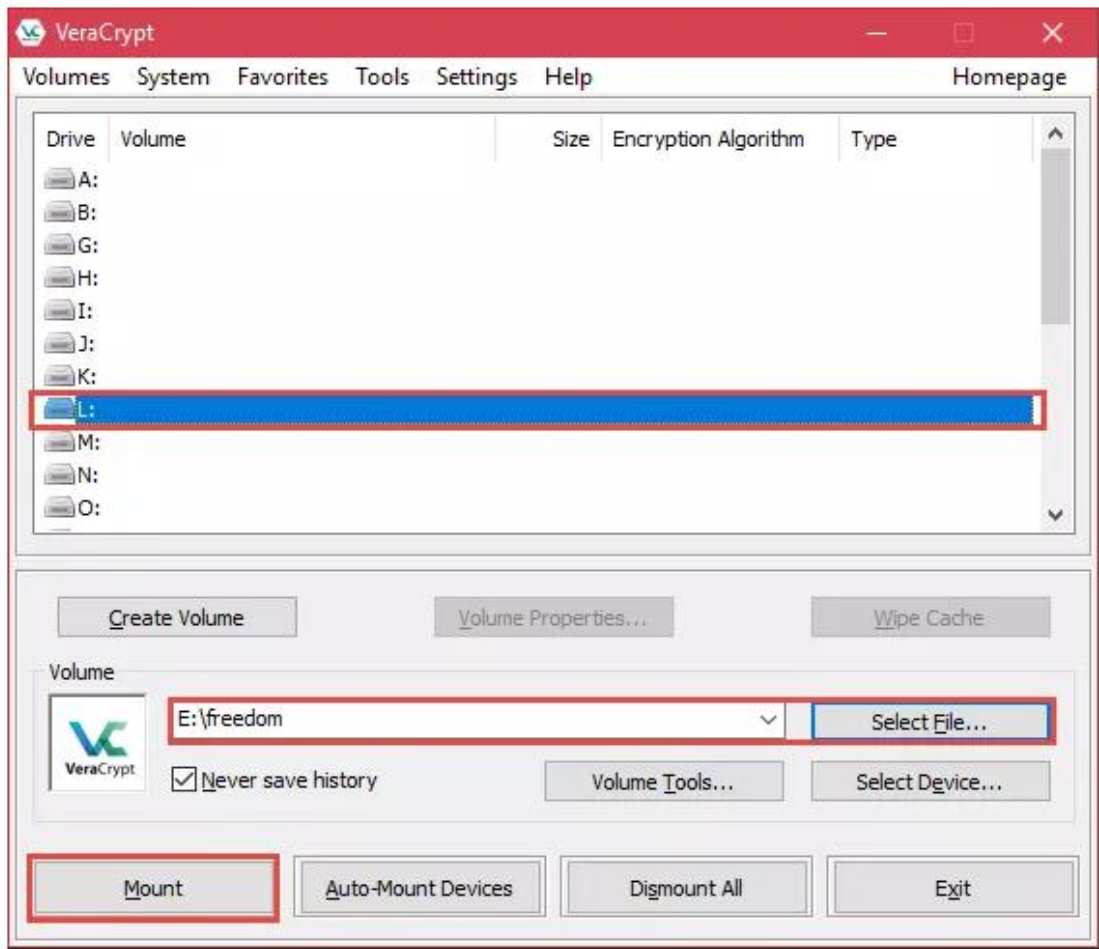


图 1.3 VeraCrypt 挂载虚拟磁盘

优点：可以将加密目录挂载到一个虚拟盘符下，方便使用，采用二级加密机制；缺点：主密钥容易被爆破，虽然这个工具也提供通过密钥文件进行解密的方式，但这种方式一是便利性不够，二是密钥文件又要想办法加密和隐藏，且又不能放在 VeraCrypt 加密区。

KeePass：主界面如图 1.4

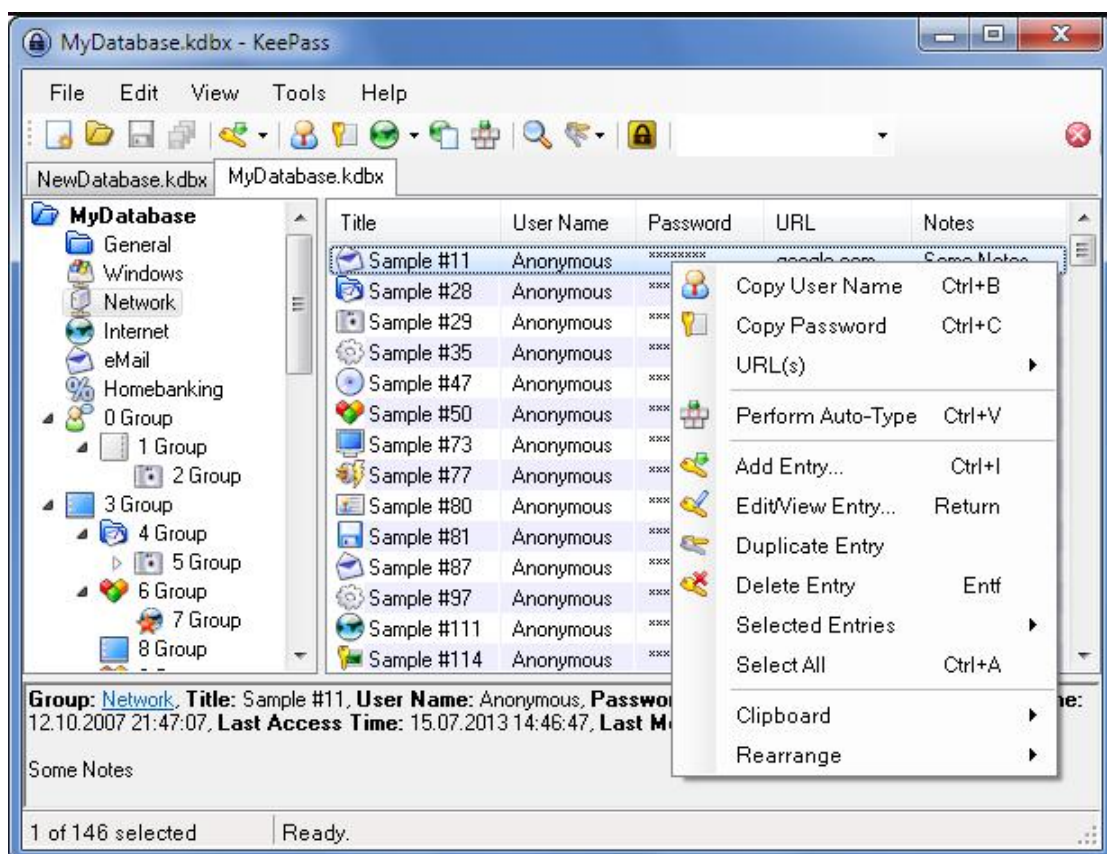


图 1.4 KeePass 主界面

通过 AES 算法来加密文件，在加密时会要求输入 AES 加密的密钥，在输入密钥后，会自动生成原文件名+扩展名+.aes 的加密文件。优点：可以使用很多自定义的加密密钥，但是加密密钥需要自己进行管理；缺点：用户管理密钥复杂，容易弄混密钥，使用时必须先解密，并不方便。

第二章 加密算法对比与选择

2.1 对称加密与非对称加密

对称加密，又称共享密钥加密，即通信双方用同一个密钥分别对信息进行加密、解密运算。^[1]。对称加密使用同一份密钥进行加密/解密，在网络应用中安全性不好，但是，在个人文件加密系统这种纯粹的本地应用中，他是比较适合。

非对称密码算法是指通信的收发双方使用具有两组密钥的密码，如 RSA 算法。一个作为公钥对外公开，另一个作为 私钥不对外公开，用来进行解密。尽管非对称密码算法需要使用两个密码增加了密码的复杂性，但是它能够有效地提高信息 安全程度，获得广泛的好评。^[2]在本应用中，由于文件按的加密解密，

都是由文件主控完成的，不需要传输，所以非对称加密算法在本程序中就不需要了。

2.2 各种对称加密算法的比较

2.2.1 DES 加密算法

DES 加密算法采用 56 位密钥对 64 位明文数据进行有效的处理，由于其算法是公开的，因此，DES 加密算法的保密性主要是由密钥的安全性决定的，其密钥是一组对称的分组密码。^[3]这是本程序中主要使用的加密算法方法之一，是一种比较基础，实用的算法，性能各方面基本满足要求，而且出错率少，可以使用。

2.2.2 3DES 加密算法

3DES 针对 DES 密码长度过短，安全性略低而进行改进，利用 DES 形成了三重数据加密。3DES 采用 3 组 64 位密钥，若第一组 和第三组密钥相同，则被称为双密钥的 3DES，即加密—解密—加密(EDE) 模式，密钥实际为 $56 \times 2 = 112$ bits。若三组密钥均不相同，那么密钥长度实际为 $56 \times 3 = 168$ bits， 安全性将会大大增强。^[4]但是本程序不会使用这种算法，因为他只是 des 的加强，但是用户密钥很难胡 hi 有这么长的密钥，更多的只是浪费；而且 3des 的软件实现比较困难，而且性能都很一般，但是文件加密中性能是很重要的，因此更不会选他。

2.2.2 IDEA 加密算法

国际数据加密算法 IDEA(International Data Encryption Algorithm)密钥长度 128 位，密钥空间是 2128，属于对称加密算法的一种，该算法具有保密性强、加密速度快的特点。^[5]在本程序中也可以使用，但更多的只是一种替补。

2.2.2 AES 加密算法

AES 算法是为了代替安全性不能满足要求的 DES 算法而选出的，主要由拓展密钥、加密模块和解密模块组成。AES 的分组长度为 128 比特，且有三种可选的密钥长度：128 比特、192 比特及 256 比特，分别对应的加密轮数为 10 轮、12 轮及 14 轮。^[6]AES 算法，性能好，安全度高，将会作为本程序的主要加密算法，当然也会提供 DES 算法和 IDEA 算法，但他们更多的只是一种补充，为了提

高破解难度而较少的使用一部分这些算法。

2.2.2 各种对称加密算法的比较

总的来说，对称加密算法 DES 比较慢，只能在安全要求不太高的场合中使用；3DES 更加慢，这是文件个人文件系统中无法忍受的；IDEA 速度快，对密钥要求不但是容易攻破，只能作为一个替补，偶尔使用一下；AES 具有速度快，有抗密码分析强度好，具有相对更好的安全性，将会被作为主力使用。

2.3 分组密码工作模式

这几种算法还要指定分组密码工作模式，主要有常用的两种：

2.3.1 ECB 电子密码本

ECB（电子密码本）是最常用的分组模式，每次加密均产生独立的密文分组，并且对其他密文分组不会产生影响，也就是相同的明文加密后产生相同的密文。比较适合数据较少的情况，对于很长的信息或者具有特定结构的信息，其大量重复的信息或固定的字符开头将给密码分析者提供大量的已知明密文对。若明文不是完整的分组，ECB 需要进行填充。^[7]

2.3.2 CBC 密文链接

CBC（密文链接）是比较常用的，明文加密前需要先和前面的密文进行异或运算，也就是相同的明文加密后产生不同的密文。由于加密算法的每次输入和本明文组没有固定的关系，因此就算有重复的明文组，加密后也看不出来了。为了配合算法的需要，有一个初始向量（iv）。与 ECB 一样有填充机制以保证完整的分组。

2.3.3 其他工作模式与总结

除了上述两种工作模式外，还有：CFB：密文反馈。OFB：输出反馈。CTR：计数器。这五种工作模式主要是密码学中算法在进行推导演算的时候所应用到的。^[8]本程序主要使用 ebc 和 cbc 模式，随机使用多种加密算法，多种分组模式，有利于加强文件系统的安全性。

2.3 算法库

对称加密算法实现是非常复杂的，而且稳定性要求是非常高的，所以，我不在自己实现对称加密算法，而是使用 openssl 库所提供的各种加密算法，对他们进行一次适配器模式的包装，使用统一的接口进行调用。

Openssl 项目是一个开放源代码的工具包，它实现了安全套接层协议 (SSLvZ/v3) 和传输层安全协议 (TLSv1)，并带有一个功效完整的、具有通用性的加密技术库。Openssl 工具包可分为三个部分：SSL 函数库、Crypto 函数库和命令行工具。SSL 函数库实现了安全套接层协议 (SSLvZ/v3) 和传输层安全协议；Crypto 函数库实现了大多数 internet 标准的加密算法；[9] 这里我们使用 openssl 的 Crypto 函数库，它实现了很多常见的加密解密算法，并将它们封装为 BIO 库（各种文件相关的操作和一些简单的算法）EVP 库（各种复杂算法的实现，兼容 bio 库），抽象出了一套通用的逻辑。

source/sink 类型的 BIO 是数据源，例如，socket BIO 和文件 BIO。而 filter BIO 就是把数据从一个 BIO 转换到另外一个 BIO 或应用接口，例如在加密 BIO 中，如果写操作，数据就会被加密，如果是读操作，数据就会被解密。[10] OpenSSL EVP (high-level cryptographic functions) 提供了丰富的密码学中的各种函数。EVP 中实现了各种对称算法、摘要算法以及签名/验签算法。EVP 函数将这些具体的算法进行了封装，并且是支持 bio 库的，他们都可以被当作一个 bio 的 filter 使用，这样就极大地简化了编程流程。

第三章 工作环境搭建

3.1 总体说明

本系统尽可能使用 posix 标准进行编写，Linux 下使用 gcc 进行编译，windows 下使用 cygwin+gcc 进行编译，对于必须使用平台相关组件的，使用宏命令进行区分编译。

整体架构采用 mvc 架构，model 与 controller 结合较为紧密，在一起相互依赖实现，view 层完全隔离，通过命令模式驱动，网络访问中使用 socket (tcp) 进行通信，本地 gui 通过 pipe 进行通信，即使没有 gui，也可以通过命令行进

行驱动。

3.2 依赖及编译/获取方式:

STL 库: c++标准库; 获取方式: c++自带。

Posix 标准库: 为跨平台准备的一些底层操作库, 如 pthread 获取方式: linux 和 cygwin 自带。

boost 库: c++准标准库, 为跨平台提供了很多高级工具, 如 boost: : filesystem, boost: : log 等, 为跨平台准备了充分的环境。获取方式: cygwin : 包管理器中有 ; linux : apt install libboost-all-dev。

gtest 库: google 的测试用例的驱动库; 获取方式: 通用: github 获取源码, cmake -DCMAKE_INSTALL_PREFIX=/usr .. & make & make install。

openssl crypt 库: 提供各种加密解密算法; 获取方式: ; cygwin : 先安装 nasm(包管理器) perl(包管理器), 然后 github 获取源码, 执行 ./Configure Cygwin-x86_64 no-asm --prefix=/usr ; linux: 直接 apt 获取。

yaml-cpp 0.6.0 : yaml 语言解析器; 获取方式: 通用: github 获取源码, 编译安装 (注意如果是静态库, 直接链接 .a 文件位置, eg; /usr/local/lib/libyaml-cpp.a, 如果是动态库, 则和其他库一样)

qt5 (qt 跨平台图像库, 性能和美观都不错); 获取方式: ; Linux: 直接安装官网安装包; Window: 直接安装官网安装包, 和 cygwin 的交互在架构处会提到。

Openssl 中使用的算法: 编码: base64; 散列: md5, sha256;

对称 DES-cbc DES-ecb AES-cbc

非对称 RSA(暂时没用到)

3.3 自动加载子模块的 cmake 脚本

构建一个自动化的 cmake 脚本, 方便多模块使用:

由于 cmake 没有可以直接使用的将某个子目录全部解析为子模块的功能, 但是我梦必定会有很多的子模块, 这样, 对导致每次新加子模块都要修改 cmakefile, 这是很麻烦的一件事, 本着没有轮子就造的精神, 我自己写了一个

自动加载子目录为子模块的脚本：

如图 3.1 的流程，在项目根目处创建一个总 cmake 文件，用于设置通用的编译选项和加载子项目。cmake 脚本中有一个 `execute_process` 命令，它可以用来在执行 cmake 脚本之前调用一些可执行程序，并将结果保存在一个变量中，有了这个功能，我们就可以实现自动加载子模块了。

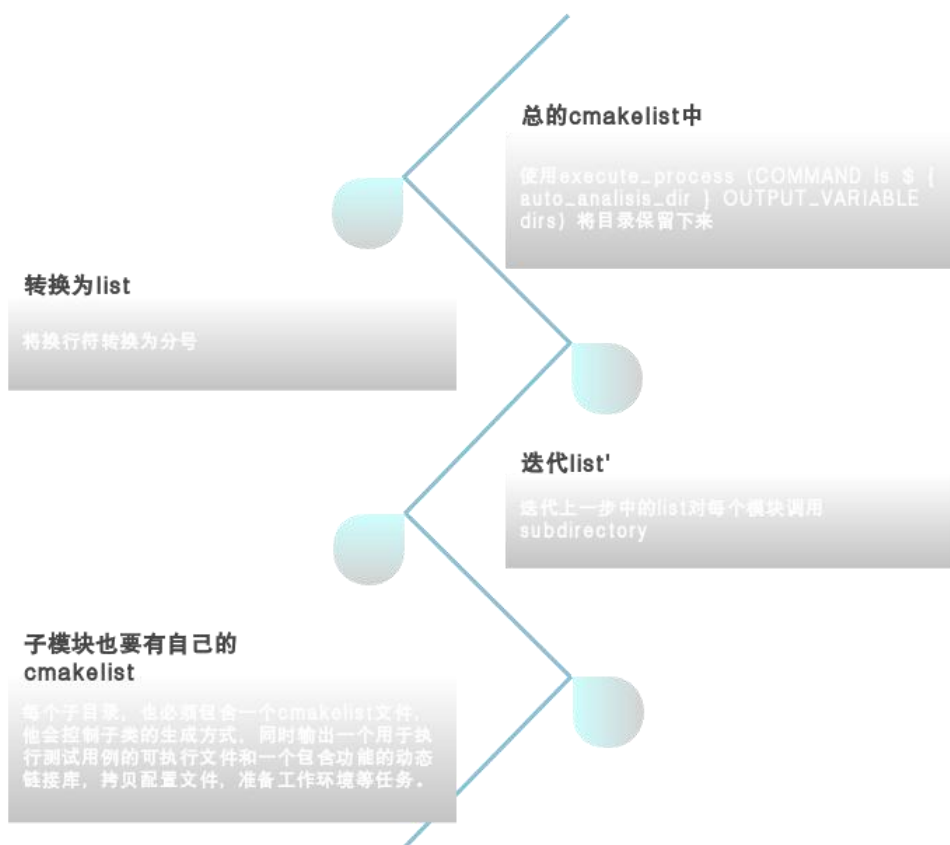


图 3.1 自动加载子模块的 cmake 脚本的流程

使用 `execute_process (COMMAND ls $ { auto_analysis_dir } OUTPUT_VARIABLE dirs)` 命令，将 `auto_analysis_dir` 下的文件列出来，注意，`ls` 命令在脚本中是通过换行符来区分文件的，但是 cmake 识别的列表是通过封号来划分的，所以使用 `string(REPLACE "\n" ";" RPLACE_LIST ${dirs})` 将换行符替换为";"，这样，我们就得到了一个列表，使用 `foreach` 命令对每一个子文件夹调用 `subdirectory`，这样，我们就可以不用修改 `CMakefile` 文件就实现自动加载子类了。

当然，对于每个子目录，如图 3.2，也必须包含一个 `cmakelist` 文件，他会控制子类的生成方式，同时输出一个用于执行测试用例的可执行文件和一个包含功能的动态链接库，拷贝配置文件，准备工作环境等任务。

```
set(common_cpp ${PROJECT_SOURCE_DIR}/common/common_include/common_funcs.cpp ${PROJECT_SOURCE_DIR}/common/common_include/log_init.cpp)
# 测试用的可执行文件
add_executable(${build_name} ${${build_name}.cpps} main/main.cpp ${common_cpp})

# 输出库 和可执行文件保持一致
add_library(${build_name}_dev SHARED ${${build_name}.cpps} ${common_cpp})

# 复制配置文件
add_custom_command(
    TARGET ${build_name}
    COMMAND echo copying files:
    COMMAND rm -rf ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/configure/
    COMMAND mkdir ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/configure/
    COMMAND cp -v ${PROJECT_SOURCE_DIR}/common/properties/boost_log_settings.ini ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/configure/
    COMMAND echo copying files is finished.
)

#boost_log 的宏
add_definitions(-DBOOST_LOG_DYN_LINK)

#动态链接库依赖
include_directories(${PROJECT_SOURCE_DIR}/src/crypto_adaptor)
include_directories(${PROJECT_SOURCE_DIR}/src/passwd_adaptor)

set(libs gtest pthread boost_log boost_log_setup boost_thread boost_system boost_filesystem crypto_adaptor_dev passwd_adaptor_dev yaml-cpp)
#可执行文件添加通用动态链接库依赖
target_link_libraries(${build_name} ${libs})
#动态链接库也要依赖
target_link_libraries(${build_name}_dev ${libs})
```

图 3.2 子模块的 `cmakelist` 关键代码

这样当新添加模块时，只要新建一个文件夹并复制子文件夹的 `cmakelist` 进行少量修改就可以同时实现输出库和测试可执行文件，十分方便。

3.4 版本控制

使用 `git` 作为版本控制工具，搭配 `gitee`（类似 `github`，只不过服务器在国内，速度快一些）进行开发。

第四章 测试驱动

单元测试是一个软件开发十分重要的过程，但是当功能较为多时，测试用例的就会变得十分繁琐；为了检测边界值，又需要复制很多重复的代码出来，十分丑陋，而且一旦功能有变，又需要修改大量的代码。这使得很多程序不进行测试开发，而大公司中有专门的测试人员负责测试（但个人项目根本不可能）。为了解决这个问题，我做出了如下设计：所有功能预先设计接口，类似的功能使用统一的接口，根据接口开发出一个测试框架通过框架加载 `yaml` 配置，使用配置调用具体的测试用例，通过如图 4.1 流程测试：

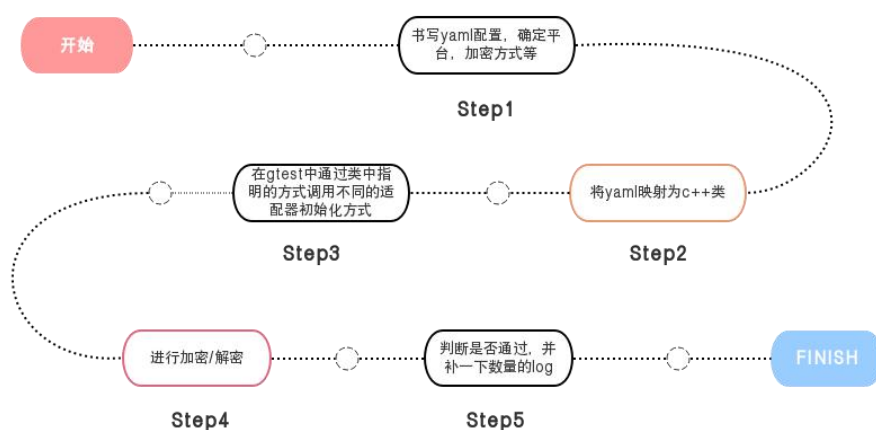


图 4.1 测试流程

这样只需要给每个接口开发一个测试用例，至于具体要测哪些值，要测多少条数据，完全由 yaml 配置文件决定，程序简洁清晰，添加新的测试用例又十分简单。加密解密适配器的部分文件如下：

```

├── hash_adaptor.cpp
├── hash_adaptor.h
├── test.hpp
├── test_main
│   ├── init_tests_yaml.cpp
│   ├── init_tests_yaml.h
│   ├── main.cpp
│   └── tests.cpp
└── work_mode.h
  
```

这里实现了各种加密算法的适配器，每个适配器由需要好多的测试用例（因为每种加密适配器同时支持文件，字符串，二进制）但是通过 adaptor_base.h 定义好接口后，就可以只写三个测试用例，然后通过 yaml 加载测试数据进来，进行测试。测试使用 Google 的 gtest 进行。一个例子如下：

测试用的 model 如图 4.2

```

name: crypt tests
nodes:
- testGroup: md5_str
  testName: hello
  platform: 0
  workNode: 1
  cryptType: 1
  source: hello
  rightResult: 5d41402abc4b2a76b9719d911017c592
- testGroup: sha256_str
  testName: hello
  platform: 0
  workNode: 1
  cryptType: 2
  source: hello
  rightResult: 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
- testGroup: md5_file
  testName: win file
  platform: 1
  workNode: 2
  cryptType: 1
  source: D:\英雄时刻\主界面、结算、签到、破产分享配置表(1).rar
  rightResult: 41812bfefde876406c8b584c8904b7ef

```

图 4.2 测试用的 model

这是测试时使用的数据，它指明了测试用例的组，测试名，可执行的平台，测试值，测试值的解释方式（文件按，base64 数据，字符串），加密方式，和预期值，然后将这个配置映射到一个 c++ 数组中，如图 4.3。

```

20 void TestNode::operator<<(const YAML::Node &node) {
21     this->testGroup = node["testGroup"].as<std::string>();
22     this->testName = node["testName"].as<std::string>();
23     this->platform = node["platform"].as<int>();
24     this->workNode = node["workNode"].as<int>();
25     this->cryptType = node["cryptType"].as<int>();
26     this->source = node["source"].as<std::string>();
27     this->rightResult = node["rightResult"].as<std::string>();
28 }
29
30 void TestNode::operator>>(YAML::Node &node) const {
31     node["testGroup"] = this->testGroup;
32     node["testName"] = this->testName;
33     node["platform"] = this->platform;
34     node["workNode"] = this->workNode;
35     node["cryptType"] = this->cryptType;
36     node["source"] = this->source;
37     node["rightResult"] = this->rightResult;
38 }

```

图 4.3 将配置映射为一个 c++ 对象

这样我们九八 yaml 配置，转换为 c++ 对象，然后通过一个测试驱动，不断地循环这个数组，就可以实现对配置进行测试：

这样，进过简单的开发，就可以实现测试用例 100% 覆盖。当然这样也引入了

一个问题：Google test 无法正确地识别到测试用例的个数，因为一个模块只用一个 test，需要我们使用 log 进行追补。

第五章 系统总体设计

文件加密/解密的管理程序，采用多级密钥管理体制，主密钥用来保护用户密钥，用户密钥用于保护个人文件的加密/解密密钥；对称密钥算法作为文件加密/解密算法，采用随机密钥加密文件，并进行相关文件/密钥资源管理。

5.1 总体架构

如图 5.1 为系统的总体加密解密流程架构。

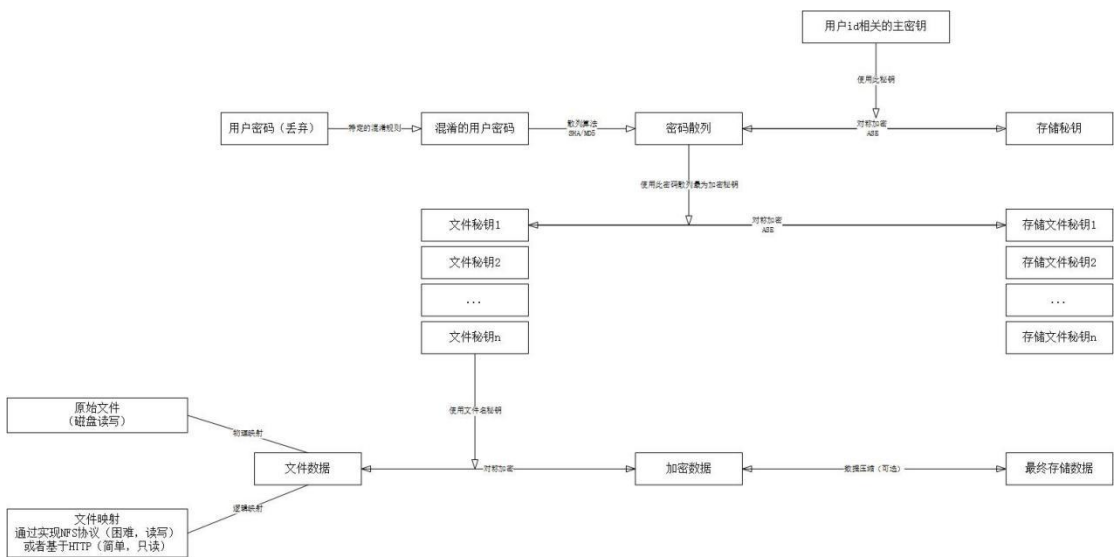


图 5.1 系统的总体加密解密架构

主密钥通过写死在动态链接库中，通过直接修改动态链接库来为每个用户提供不同的主密钥；用户密钥通过查表加排序，并和主密钥用掩码进行互补保持，使用主密钥加密的 yaml 文件按存储相关信息；文件密钥使用用户密钥加密；文件使用文件密钥进行加密。

5.2 总体模块说明

主要采用适配器模式, mvc 模式, 组件模式 (unity3d 类似的解耦设计思想), 单例模式, 观察者模式, 命令模式等设计思想, 通过各种控制器与组件模式进行整合。

5.2.1 项目主目录

—— CMakeLists.txt	: 总的 cmakeList
—— common	: 通用的工具, 配置文件, 日志适配器等
—— LICENSE	: 许可协议 (BSD) 和其他依赖的协议
—— README.en.md	: reamde 的英文版
—— README.md	: readme 的中文版
—— gui	: 通过命令模式驱动的 qt gui 的实现
—— src	: 各个功能模块的实现

5.2.2 common: 包含通用的工具, 配置文件, 日志适配器等

—— common/common_include	: 日志适配器, 常用方法, 版本号等
—— common/properties	: 配置文件
—— common/temlpate	: cmakeList 的模板

5.2.3 src: 各个功能模块动态链接库和测试可执行文件的实现

—— src/change_rodata	: 修改已编译动态链接库中的主密钥
—— src/crypto_adaptor	: 加密算法适配器
—— src/file_coltroller	: 文件管理器的逻辑实现
—— src/main_controller	: 主控, 用于协调各个模块协作
—— src/main_passwd	: 主密钥的管理器
—— src/passwd_adaptor	: 用户密钥与文件密钥的管理器
—— src/web_adaptor	: 简单的网络实时解密浏览实现

5.2.4 gui: 通过命令模式驱动的 qt gui 的实现

—— personal_file_gui_qt	: qt + qml 写的一个 gui
-------------------------	---------------------

5.3 生成的可执行文件说明

——	configure	: 拷贝的配置文件
	—— boost_log_settings.ini	: boostlog 的配置文件
——	cygchange_rodata_dev.dll	: 初始化时修改主密钥的链接库
——	cygcrypto_adaptor_dev.dll	: 加密适配器 (使用 openssl)
——	cygfile_controller_dev.dll	: 文件管理器, 管理文件表
——	cygmain_passwd_dev.dll	: 主密钥的存储的 dll 库
——	cygpasswd_adaptor_dev.dll	: 用户密钥, 文件密钥的适配器
——	cygweb_adaptor_dev.dll	: 简单的网络浏览适配器
——	filedata	: 加密文件数据
	—— data.7AWUCj	
	—— data.B4WSQL	
——	log	: 输出的日志
——	main_controller.exe	: 主控, 通过 pipe 进行获取命令
——	main_controller_cli.exe	: 通过命令行驱动的 pipe 客户端
——	personal_file_gui_qt.exe	: qt 写的 gui 驱动的 pipe 客户端
——	sysdata	: 系统数据
	—— 2.filePasswd.data	: 用户 2 的文件密钥
	—— 3.filePasswd.data	: 用户 3 的文件密钥
	—— 3.index.data	: 用户 3 的文件目录
——	init.data	: 是否初始化的记录
——	pipe.data	: 主控启动时的随机 pipe 位置记录
	—— userKey.data	: 获取用户密钥的关键数据

当然这只是本系统生成的动态链接库和可执行文件, 其他以来的库就不再这里列举了。

第六章 各个模块的详细说明与实现方式

6.1. common/common_include 通用方法和头文件

common/common_include/

—— common_configure.h.in	: cmake 的版本控制头文件
—— common_funcs.cpp	: 通用方法
—— common_funcs.h	: 通用方法的定义
—— common_includes.h	
—— log_init.cpp	: log 的适配器实现
—— log_init.h	: log 的适配器定义
—— README.md	

common_funcs.h: 通用方法, 包含几个内存安全的内存拷贝和读写超时检测
size_t memncpy(void *__restrict dest, size_t dest_len, const void
*__restrict src, size_t src_len);由于 memcpy 不会检测是否越界, 这样会导致很可能写坏堆栈, 造成很对安全问题, 这里对 memcpy 进行了一次包装, 这样就不会写坏堆栈了, 类似 strncpy 的实现, 额外传入了 dest_len, 最多只拷贝这个长度的数据, 并返回实际拷贝的长度, 方便检测是否越界写了。

#define MEMNCPY(dest, dest_len, src, src_len) memcpy(dest, src, (src_len < dest_len) ? src_len : dest_len)通过宏定义实现的 memncpy, 由于函数调用还是比较耗费性能的, 而 inline 也并不是一定会起作用的, 但是这个函数一定会被频繁调用, 所以通过宏定义实现, 减少函数调用开销。

int test_read_timeout(int fd, long wait_sec);检测读超时的函数 (并不进行读操作)

int test_write_timeout(int fd, long wait_sec);检测写超时的函数 (并不进行读操作)

log_init.h: log 的适配器定义

如图 6.1 通过适配器模式对 boost.log 的封装, 方便在不修改代码的情况下更换 log 库。如刚开始时是使用 log4cplus, 但是由于它在 cygwin 环境下无法通过编译, 而跟换为了 boost.log, 却不用修改任何代码。

```

extern src::severity_channel_logger<severity_level, std::string> logger1;
extern src::severity_channel_logger<severity_level, std::string> logger2;

#define TRACE(msg) BOOST_LOG_SEV(logger1, trace)<<msg<<endl;  [ "<<__FILE__<<": "<<__LINE__<<" " "<<__PRETTY_FUNCTION__<<";
#define DEBUG(msg) BOOST_LOG_SEV(logger1, debug)<<msg<<endl;  [ "<<__FILE__<<": "<<__LINE__<<" " "<<__PRETTY_FUNCTION__<<";
#define INFO(msg) BOOST_LOG_SEV(logger1, info)<<msg<<endl;    [ "<<__FILE__<<": "<<__LINE__<<" " "<<__PRETTY_FUNCTION__<<";
#define WARNING(msg) BOOST_LOG_SEV(logger1, warning)<<msg<<endl;  [ "<<__FILE__<<": "<<__LINE__<<" " "<<__PRETTY_FUNCTION__<<";
#define ERROR_LOG(msg) BOOST_LOG_SEV(logger1, error)<<msg<<endl;  [ "<<__FILE__<<": "<<__LINE__<<" " "<<__PRETTY_FUNCTION__<<";
//只是一个参考，调用errno检查错误
#define PERROR(msg) BOOST_LOG_SEV(logger1, error)<<msg<<endl;  [ "<<__reason maybe : "<<strerror(errno)<<" " "<<__FILE__<<": "<<__LINE__<<" " "<<__PRETTY_FUNCTION__<<";
#define FATAL(msg) BOOST_LOG_SEV(logger1, fatal)<<msg<<endl;  [ "<<__FILE__<<": "<<__LINE__<<" " "<<__PRETTY_FUNCTION__<<";
////// 原始输出
#define RAW_COUT(msg) std::cout<<msg;
#define RAW_CLINE(msg) std::cout<<msg<<endl;
#define RAW_PRINTP(fmt, args...) printf(fmt, ##args);
#define RAW_PLINE(fmt, args...) printf(fmt, ##args);printf("\n");

```

图 6.1 日志封装提供的接口

对常用的所有可能的输出都进行了包装，通过宏定义实现，也不会耗费性能，并且剥离了业务逻辑也日志框架，减少了对业务的侵入性，当由于各种原因要更换日志模块时，只需要在这里进行修改就好，完全不用改动业务代码。

6.2 src/crypto_adaptor: 加密解密算法适配器

—— adaptor_base.h	: 加密库适配器基类
—— DES_func_adaptor.cpp	: des 适配器
—— DES_func_adaptor.h	: des 适配器
—— AES_func_adaptor.cpp	: AES 适配器
—— AES_func_adaptor.h	: AES 适配器
—— base64_adaptor.cpp	: base64 适配器
—— base64_adaptor.h	: base64 适配器
—— CMakeLists.txt	
—— hash_adaptor.cpp	: md5, sha256 适配器
—— hash_adaptor.h	: md5, sha256 适配器
—— test_main	
—— init_tests_yaml.cpp	: 测试配置加载
—— init_tests_yaml.h	: 测试配置加载
—— main.cpp	: 测试用例驱动的主函数
—— tests.cpp	: 测试用例的定义
—— work_mode.h	: 工作模式如平台，加密方法等

对 openssl crypt 库进行适配器模式封装。如图 6.2 各种加密适配器都继承自 adaptor_base:


```

55
56 class AdaptorBase {
57     protected:
58         WorkMode workMode {ADAPTOR_MODE_MAX};
59         CryptType cryptType {ADAPTOR_MAX};
60         const std::string value {};
61         size_t datalen {};
62         const char *data {};
63         int status {-1};
64         std::string strResult("");
65
66         __uintmax_t totalSize {};
67         __uintmax_t finishSize {};
68
69     public:
70         virtual size_t getTotalSize() {
71             return totalSize;
72         }
73
74         virtual size_t getFinishSize() {
75             return finishSize;
76         }
77
78     public:
79         AdaptorBase() {
80             workMode = ADAPTOR_MODE_NONE;
81         }
82
83         AdaptorBase(WorkMode eMode, const std::string &strValue) : workMode(eMode), value(strValue) {
84         }
85
86         AdaptorBase(const char *cstrData, size_t ilen) : datalen(ilen), data(cstrData) {
87             workMode = ADAPTOR_MODE_DATA;
88         }
89
90     public:
91         virtual int isWrong() {
92             return status;
93         }
94 }

```

图 6.2 adaptor_base 提供的接口

这里只提供通用的接口，和为观察者模式准备的一些接口，至于具体的加密解密方法的定义，则由各个实现自己定义，应为各个库的接口都不相同。但是都遵循图 6.3 流程：



图 6.3 加密解密算法流程

初始化：构造适配器，指定加密/解密函数，指定加密源和类型（文件，base64 数据，文本等）

编码解码：调用 encode 或 decode 函数，结果会被暂存在 result 中

判断成功性：encode 和 decode 会返回适配器本身的指针，直接在这里调用 isWrong 函数。

获取结果：调用 getResult 函数，获取结果

释放资源：释放资源（如果有的话）

这样，就把各种复杂的加密解密逻辑概括为三句代码，使用非常方便。

如图 6.4，以 hash_adaptor 为例：Hash_adaptor 只需要一个 encode (CryptType eType)；传入算法的 enum 类型即可完成 hash 函数。，至于到底时文件，数据，还是文本，则是在初始化时指明的。而各种函数的实现，则是通过对 openssl 的封装，如对 MD5 的封装，可以方便地实现 MD5。

```
int HashAdaptor::md5_file() {
    namespace bf=boost::filesystem;
    bf::path file(value);
    if ((!bf::exists(file)) || (!bf::is_regular(file))) {
        ERROR_LOG("file not exists : " << value)
        return -1;
    }

    char fileBuff[FILE_READ_BUFF] {};
    unsigned char md[MD5_DIGEST_LENGTH];
    bf::ifstream inFile(file);
    totalSize = bf::file_size(file);
    MD5_CTX ctx {};
    MD5_Init(&ctx);
    while (!inFile.eof()) {
        inFile.read(fileBuff, sizeof(fileBuff));
        MD5_Update(&ctx, fileBuff, inFile.gcount());
        finishSize+=inFile.gcount();
    }
    MD5_Final(md, &ctx);
    this->strResult = printHash(md, MD5_DIGEST_LENGTH);
    return 0;
}
```

图 6.4，hash_adaptor 提供的 MD5 文件算法

通过初始化 MD5 控制器→不断读入一个文件→然后调用 update，同时更新观察者需要的已处理的文件大小→结束并刷新→输出结果；这个流程进行求解。

而对于 des，aes 库，如图 6.5，则会变得复杂的多，因为不仅涉及到加密解

密，还有对文件，二进制，甚至网络中的文件描述符进行操作。

```
AESAdapter *encodeDataToData(size_t len, char *outputData, size_t &outlen);

AESAdapter *encodeFileToFile(std::string fileName);

AESAdapter *encodeStringToString();

AESAdapter *decodeDataToData(size_t len, char *outputData, size_t &outlen);

AESAdapter *decodeFileToFile(std::string fileName);

AESAdapter *decodeStringToString();

int aesDataToData(std::string keyStr, const char *inData, size_t inlen, char *outData, size_t &outlen, int type);

int aesFileToFile(std::string keyStr, std::string inFile, std::string outFile, int type, size_t &curr_size);

int aesFdToFd(std::string keyStr, int inFd, int outFd, int type, size_t &curr_size);
```

图 6.5 aes 适配器提供的接口

对称加密算法的设计的就比较复杂，因为它要面对最复杂的业务。所以他提供了加密解密文件，数据，字符串，甚至文件描述符的功能。

如图 6.6，对文件描述符的操作为例：

```
int aesFdToFd(std::string keyStr, int inFd, int outFd, int type, size_t &curr_size) {
```

图 6.6，对文件描述符的操作函数接口

参数说明：KeyStr：密钥，通过文件密钥控制器获取；inFd：待加密/解密的文件描述符；outFd：输出的文件按描述符，支持文件，socket 等，可以实现不用解密而通过网络浏览；type：加密（0）或者解密（1）；curr_size：当前已经处理的大小，用于展示进度。

通过 openssl 提供的 bio 库，将网络的输出定义为一个 bio 文件，然后讲之与一个 des_cbc 的过滤器相互串联，同时，指定密钥，初始化方式，还有是加密还是解密，这样写入到过滤器中的数据就会被输出到文件描述符中，通过不断的读入文件->输出到过滤器->更新观察者需要的数据这一流程，就实现了对网络的加密解密输出，方便用户在无须解密的情况下，进行在线的数据的读取，甚至通过 wifi 进行共享。

至于加密解密方法，openssl 的 EVP 库对这些复杂的加密算法进行了非常好的封装，同时还支持 BIO。通过初始化一个文件 bio->将输出文件绑定到 bio 上

->初始化一个过滤器->将加密解密算法绑定到过滤器上->将输出文件绑定到过滤器之后->向过滤器写入文件->结束并刷新这一流程,就可以实现对任意加密算法的快捷使用,要修改加密算法,只需要修改如图 6.7 中这一个地方就好,非常方便。

```
if (!EVP_CipherInit_ex(ctx, EVP_des_cbc(), NULL, key, iv, type)) {  
    ERROR_LOG("EVP_CipherInit_ex Error\n");  
    return -1;  
}
```




图 6.7 解密算法的指定

6.3 src/file_coltroller: 文件管理器的逻辑实现

├── CMakeLists.txt	: 子模块 cmake
├── file_controller_adaptor.cpp	: 文件控制器主控适配器
├── file_controller_adaptor.h	: 文件控制器主控适配器
├── file_controller.cpp	: 文件与文件表实现
├── file_controller.h	: 文件与文件表实现
└── main	
├── init_tests_yaml.cpp	: 测试用例加载器
├── init_tests_yaml.h	: 测试用例加载器
├── main.cpp	: 测试主类
└── tests.cpp	: 测试驱动

设计思路与原因:

首先: 文件树的方式有非常复杂的层次关系, 对于大量的文件进行分层管理效率高, 但是复杂的层次关系导致他非常脆弱, 一点文件错误可能导致很多文件索引不到, 要对这种情况进行处理需要非常复杂的逻辑, 即使是现代的文件系统, 很多也处理不好, 而单层的文件结构有非常好的抗错误能力, 但是文件多的时候性能不行, 考虑到个人文件系统不会有太多的文件, 但是稳定性非常重要, 所以最后选择了单层文件模式, 因此采取了单层文件结构+全路径名的方式实现而非文件树结构的方式。

对于文件表的存储实现方式, 我有三种思路: 通过 map+yaml/json 存储, 通

过 map+new(dest) 直接 dump 内存实现, 通过 sqlite 实现。

map+yaml/json 存储: 需要写一些打包函数, 存储, 加载会慢一些; 但是稳定性好;

map+new(dest) 直接 dump 内存实现: 通过 c++<new>库, 在指定的 buff 中 new 出 map, 然后对这个 map 进行操作, 存储和加载则直接对这一段 buff 进行操作, 实现简单, 但是用户可能无法进行迁移, 由 32 位系统换成 64 位系统, 或者进行系统间迁移都会存在问题。而且, 直接使用内存映射, 会导致一旦某个位置出现文件错误, 那么可能会造成整个列表出现偏移, 那么所有数据都会丢失。而通过加密的字符串的形式存储, 即使出现了文件错误, 也只是导致他本身解析出错, 最多影响邻居, 不会有大的灾难。

sqlite: 如果把前面两种实现方式看作 redis, 那么 sqlite 可以看做传统 sql, 由于单层化的模式, 它的性能可能存在问题; 每次查找都要去磁盘加载数据库, 速度堪忧, 尤其时当有其他线程在进行加密/解密操作时, 会大量占用磁盘速度, 这时, 数据库进行更新会变得非常慢; 而且 sqlite 开源版的加密是对数据进行加密, 但是表结构是公开的, 当用户使用较为简单的加密方式, 如 des-cbc 时, 这是非常危险的, 而加密表结构的版本时要商业收费的, 这会带来很大的安全隐患。因此在开发一段时间后只能放弃, 换用其他方式进行持久化存储。

考虑到上述理由, 我选择了 map+yaml 存储, 使用 aes 算法和用户密钥 (本设计中最安全的密钥) 进行加密。

具体实现:

文件表采用类似 unix 方式, 但只保留 owner (r,w) other (r,w), 文件列表通过 yaml 加密存储, 文件扁平化设计, 只有一层, 通过类似 linux 的 iNode 方式管理, 通过路径虚拟出文件夹。

因次选择自己早轮子, 考虑到 yaml 的 c++库比我所接触到的 json 的 c++库使用要方便很多, 所以这里使用 yaml-cpp 作为序列化方式, 采用一张 hashmap 在内存中存储数据, 实现快速查找, 通过 yaml 序列化这张表, 然后进行加密, 本地存储的思路进行管理。

std::string encName: 文件在显示时使用的名字; int encMode: 文件加密

的方式，如 des-cbc，des-ebc 等；std::string encCryptFileName：如果时加密文件，则它在磁盘中存储的文件名或索引文件名；int encFileKey：文件的控制权限位标记（r，w）；int encFileKey：用户的文件密钥 id；int accessBit：访问权限。

这些就是全部要保存的数据了，比较简单，不涉及修改时间，访问时间之类的复杂逻辑，只是记录最基本的信息，而其他的信息，则使用 encCryptFileName 所指向的文件的数据。然后，如图 6.8，通过 yaml 将其序列化为一段字符转，作为每一条记录的原始数据，或者从解密的 yaml 中加载进内存。

```
12
13 void EncINode::operator<<(const YAML::Node &node) {
14     this->encFileType = emFileType(node["encFileType"].as<int>());
15     this->encName = node["encName"].as<std::string>();
16     this->encMode = node["encMode"].as<int>();
17     this->encCryptFileName = node["encCryptFileName"].as<std::string>();
18     this->encFileKey = node["encFileKey"].as<int>();
19     this->accessBit = node["accessBit"].as<int>();
20 }
21
22 void EncINode::operator>>(YAML::Node &node) const {
23     int type = this->encFileType;
24     node["encFileType"] = type;
25     node["encName"] = this->encName;
26     node["encMode"] = this->encMode;
27     node["encCryptFileName"] = this->encCryptFileName;
28     node["encFileKey"] = this->encFileKey;
29     node["accessBit"] = this->accessBit;
30 }
```

图 6.8 yaml 和 c++之间的映射

这样每一条元数据的加载与存储的问题就解决了。当然，这个元数据是不负责加密与解密的，因为它只是一条记录，可以是文件，也可以是文件夹，甚至可以是一个没有加密的文件，所以它只是一条记录，加密解密不应该由他负责。

然后是对这个元数据的管理，时通过一张 map 表进行管理的，同样使用 yaml 进行存储或者读取，通过 yaml 的 push_back 函数，来记录表。

当然，这里也不会做加密解密操作，打的作用只是对上面的 iNode 进行管理，通过一个 map 进行管理，c++stl 的 map 使用了红黑树，查找效率比较高，而且也实现了对整张表的序列化，这样可以直接在内从中进行修改，然后在后台同步回文件，速度很快。

至于文件夹，这里不做强制要求，可以加也可以不加，因为实现了
std::map<std::string, EncINode> * selectFileWithDirByName(std::string
name, std::map<std::string, EncINode> *result);这个方法，它会找出某个文

文件夹下的所有文件，如果某个文件深于两层，那么只会把它的父目录添加进 result 中，这样，在表中完全不用理会文件夹，而是在具体的拉某个目录时，再计算出文件夹，方便其他逻辑操作。

通过这个 EncITable，完全实现了增删改查，当然这只是对记录，文件的加密解密操作不由他负责。

如图 6.9，就是它的持有者了，这个持有者将通过组件模式，加载各种功能，通过单例模式保持全局唯一，实现文件操作的全部功能。

```
16 #define DIR_FILE "../filedata/"
17
18 class MainFileController {
19 private:
20     MainFileController() = default;
21
22 public:
23     MainFileController(MainFileController &) = delete;
24
25     MainFileController(MainFileController &&) = delete;
26
27 private:
28     static std::mutex lock;
29 public:
30     static MainFileController *Instance;
31 public:
32     EncITable table {};
33     int userID {};
```

图 6.9 文件主控的构造器

首先，关掉拷贝，移动构造函数；MainFileController(MainFileController &) = delete; MainFileController(MainFileController &&) = delete; 这是 c++ 11 的新特性。

然后让默认构造函数为 private 的 MainFileController() = default; 这也是 c++ 11 的新特性。阻止它的拷贝，移动和构造，全方面防止手误导致单例模式被打破，使用饿汉模式进行单例实现（这里不需要懒汉模式的懒加载，应为他们启动时都是要构造的，至于加载表，则是推迟到了用户登录之后，饿汉模式不会影响性能，还不会有线程安全问题）

然后保持一个 static 的指针，和一个全局锁，当进行操作时必须加锁，这里是为多线程做准备，因为 gui 中一定会用到多线程操作，而除了 dump 和加载表，其他操作都是非常快的，这个消耗完全可以接受。

然后通过组件模式让他拥有文件表的增删改查的功能，并对所有文件相关的功能进行一次包装，对外之暴露他的接口。

如图 6.10，通过析构函数自动释放锁：

```
class LockGround {  
    std::mutex *mlock;  
public:  
    LockGround(std::mutex *_lock) : mlock(_lock) {  
        mlock->lock();  
    }  
    ~LockGround() {  
        mlock->unlock();  
    }  
};
```

图 6.10 通过析构函数自动释放锁

LockGround 为通过析构函数实现的一个无论如何都会释放锁的 ground，防止造成死锁，在 lockGround 被析构时调用释放锁的函数，这样，即使发生错误退出了本函数，也一定会释放锁的。

文件管理模块对外提供的所有需要对文件进行操作的接口，大部分都是对文件表（map）的简单操作，但是有几个特殊的功能：

encAndInsertFile：加密文件并且添加到文件表中；decToFile：解密一个文件到指定的位置；decToFd：解密一个文件到指定的文件描述符（为网络访问做铺垫）；这三个函数都是对前面实现的各种功能的调用。

getRandFileName：获取一个不会重复的文件名（一定数量先）这里使用 mktemp 函数；使用“./filedata/data.XXXXXX”作为模板，会随机出一个不会重复的文件名，将 XXXXXX 替换为目标文件名。

存盘/加载：

当要将表写回磁盘时，如图 6.11，先将其序列化为一段 yaml 字符串，然后使用用户密钥（后文再说它的实现方式）进行加密，最终写回磁盘，加载同理。

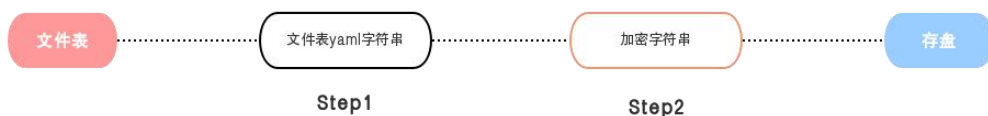


图 6.11 文件表存盘逻辑

6.4 修改动态链接库的主密钥：

src/main_passwd：主密钥的管理器和 src/change_rodata：修改已编译动态链接库中的主密钥，它们共同构成了主密钥，按照如图 6.12 的流程进行主密钥运行时修改的实现。

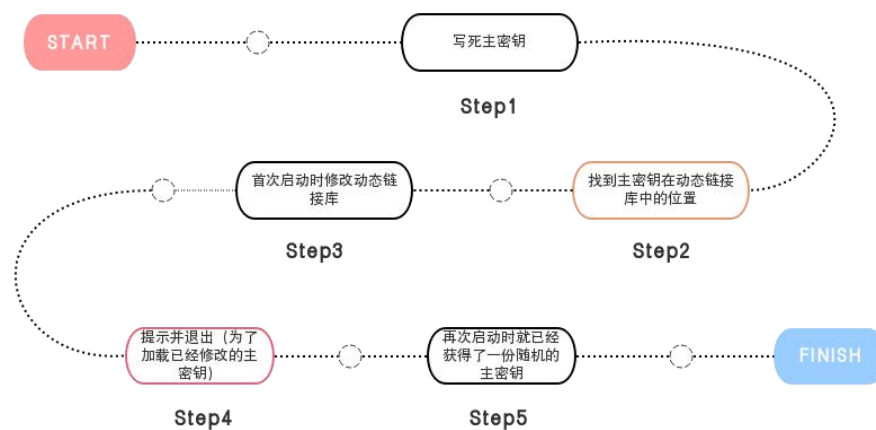


图 6.12 主密钥运行时修改的实现

- (1) 安装时创建一个全局密钥
- (2) 全局密钥在代码中写死在动态链接库中
- (3) (初次启动) 时修改该动态链接库中的密钥文本
- (4) 这样就获得了一份直接写在动态链接库中的随机公用密钥

实现方法为：c++内存模型中.data 保存有初始化的全局变量，.rodata 保存全局常量，只要在源码中声明一个很长的字符串，那么它就会被保存在一个固定的地址，可以 mmap 该库，直接修改对应位置。

如图 6.13，在 src/main_passwd，主密钥的管理器中，非常简单的记录了一段字符串做为主密钥：

```
const char main_passwd[40]="df54w8df4u7t89as54flhhkxert789s4gs6ld";

const char *getMainPasswd() {
    return main_passwd;
}
```

图 6.13 主密钥的记录

它非常简单，就是返回一个写死的字符串，但是这是不能作为主密钥的，因为这意味着一旦一个用户的主密钥被枚举破解，那么所用用户的主密钥都将被破解。所以必须要能对主密钥在编译完成之后还能够进行修改。

如图 6.14，完成了修改已编译动态链接库中的主密钥。

```
int rand_rodata() {
    namespace bf=boost::filesystem;
    bf::path fPath = SO_FILE;
    if (!bf::exists(fPath))
        || (!bf::is_regular(fPath)) {
        ERROR_LOG("err file : " << fPath.string())
        return -1;
    }

    int fd = open(fPath.string().c_str(), O_RDWR);
    size_t len = bf::file_size(fPath);
    char *ptr = (char *) mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    char *dest = (ptr + POS);
    printf("get: %s\n", dest);
    for (int i = 0; i < 30; ++i) {
        *(dest + i) = randChar();
        DEBUG("inline for: " << dest);
    }
    DEBUG("set: " << dest);
    if (munmap(ptr, len)) {
        ERROR_LOG("err when mmap")
        return -1;
    }
    return 0;
}
```

图 6.14 修改已编译动态链接库中的主密钥

mmap() 系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以像访问普通内存一样对文件进行访问，不必再调用 read(), write() 等操作。实际上，mmap() 系统调用并不是完全为了用于共享内存而设计的。它本身提供了不同于一般对普通文件的访问方式，进程可以像读写内存一样对普通文件的操作。^[11]使用 mmap 函数将这个 so 文件映射到内存中来，而非调用，这样我们只要能知道主密钥在动态链接库中的位置，就可以做到在用户使用时为他随机一份主密钥出来。

首先得找到主密钥在动态链接库中的偏移：

Linux 下使用 objdump -tT 命令，可以查看一个动态链接库的符号表：

libmain_passwd_dev.so: 文件格式 elf64-x86-64

SYMBOL TABLE:

00000000000001c8	1	d	.note.gnu.build-id
------------------	---	---	--------------------

```

0000000000000000 .note.gnu.build-id
00000000000001f0 1      d  .gnu.hash 0000000000000000 .gnu.hash
.....
00000000000201018 1      d  .data 0000000000000000 .data
00000000000201020 1      d  .bss  0000000000000000 .bss
.....
0000000000000000 1      df *ABS* 0000000000000000 main_passwd.cpp
00000000000005c0 1      0 .rodata 0000000000000028
_ZL11main_passwd
.....

```

可以看到 .rodata 在 0x5c0 位置，转换为十进制就是 1472，而我们只有一个全局常量，那么它记录的就一定是主密钥：

使用一个测试 main 进行查看：直接将 so 映射到内存中了，然后对这个内存加上 1472 的偏移量，就会指向我们的主密钥了。我们可以通过 GDB 查看这段字符是不是我们的密钥：

```
(gdb) p dest
```

```
$2 = 0x7f19c61d05c0 "df54w8df4u7t89as54flhhksert789s4gs6ld"
```

这个和我们的主密钥是一致的，因此可以确定这里就是我们的主密钥，然后就像操作数组一样对主密钥进行修改，为了清楚一点，这里将目标改为了字母 o，只是为了看每次改到哪里，具体实现中会改为随机字符：

```

[2019-04-10 22:28:28.390500] [0x00007f603b238b80] [debug] inline
for:of54w8df4u7t89as54flhhksert789s4gs6ld]
[/home/tao/linux_based_projects/personal_file_protector/src/change_ro
data/main.cpp:36 int main(int, char**)
.....
[2019-04-10 22:28:28.391229] [0x0 00 07f603 b23 8b 80] [deb ug]
set:o 000000 00000 00000 000 0000000 00os4 gs6ld]
[/home/tao/linux_based_projects/personal_file_protector/src/change_ro
data/main.cpp:38 int main(int, char**)

```

通过不断的日志我们确实可以看到主密钥一步步被修改，最后通过 munmap(ptr, len) 函数解除映射，将数据写回磁盘，这样改写会动态链接库，再次检查，发现库中的数据已经被修改：

```
(gdb) p dest
```

```
$1 = 0x7f28ca 5c05c0 "oooooo oooo oooooooooo oooooooooo oooos4 gs6ld"
```

在使用 objdump 检查库函数符号表发现无误。证明修改后的库可以使用。至于为什么不全修改，这里最好保留一两位 buff，因为一旦修改超了，那么这个 so 文件将无法使用。

而在 Windows 下，动态链接库并没有这么规整的编码方式，Windows 下色 dll 库会进行一些对齐操作，导致这一招不能使用，但是，如图 6.15，我们可以借助 Binary Viewer 这个工具直接查看二进制，并且搜索我们的主密钥，就可以找到对应的位置了：

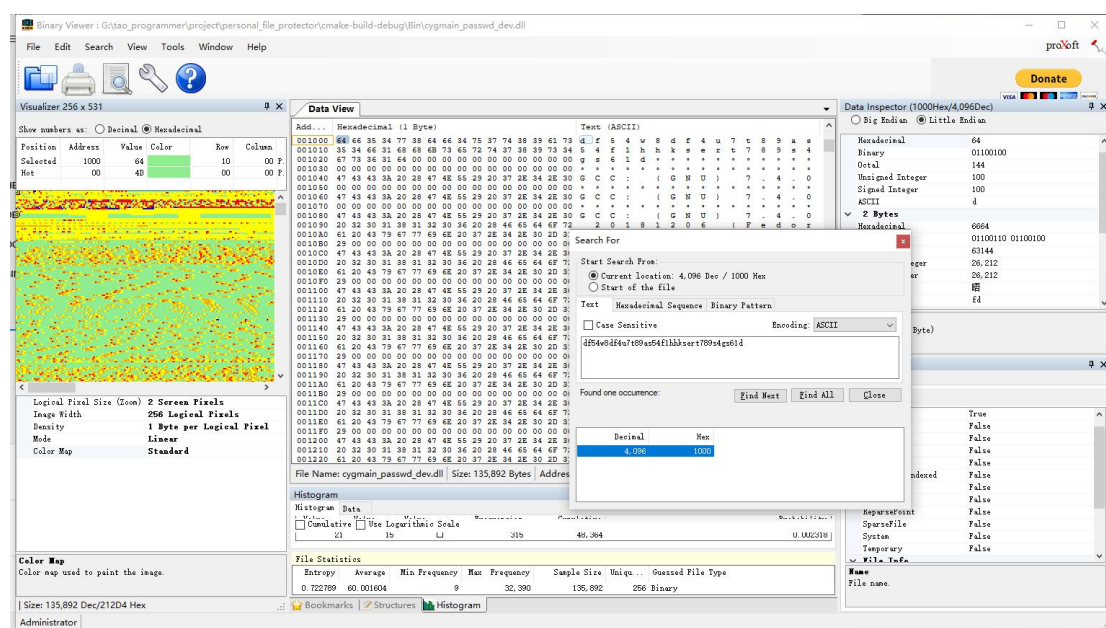


图 6.15 Binary Viewer 查找主密钥

可以看到，主密钥被对齐到了 0x1000，即 4096 的位置，这是比较奇怪的，但是通过检查发现还是可以使用的，如图 6.16，图 6.17，可以正常对主密钥进行修改。

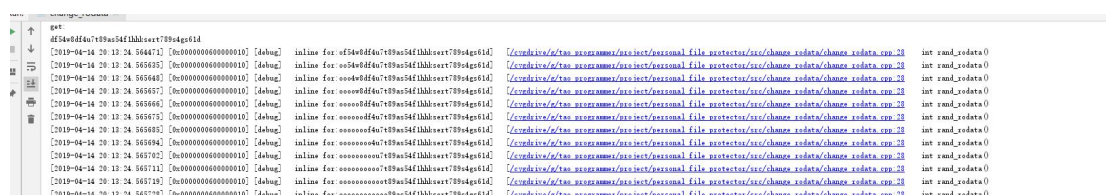


图 6.16 修改动态链接库中的主密钥过程

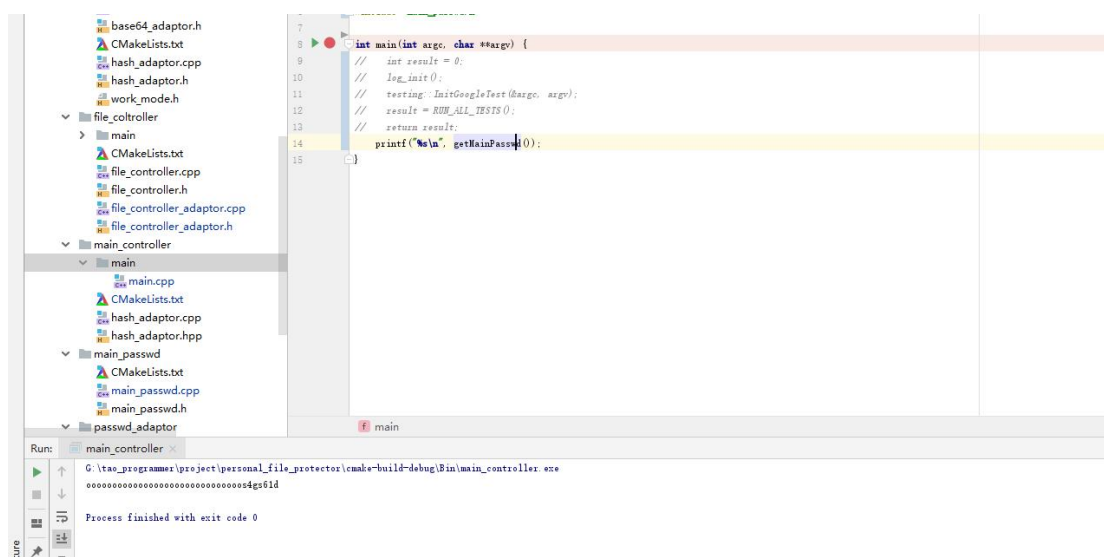


图 6.17 验证可以正常使用

我们将 so 库名修改为 dll 库名，并将偏移指定为 4096，重复之前的操作，发现同样时可以使用的。

6.5 src/passwd_adaptor: 用户密钥与文件密钥的管理器

这部分主要分为文件密钥与用户密钥。他们对外提供两个非常简单的接口：

```
std::string get_usr_key(int para);
```

```
std::string get_file_key(int user,int para);
```

通过用户 id（内部实现，用户无感）或与用户密钥，通过用户 id 和文件密钥 id 或与文件密钥。

用户名和密码的 hash 将通过主密钥加密，通过 yaml 存储，这个前文已将讲过类似的了这里不在细说。登录时用户输入用户名和密码，但是只是和密码的 hash 进行比较，即使有人破解了主密钥，也完全不能拿到其他人的密码，只拿到 hash 时没用的。

而文件密钥也是类似的情况，通过一张 yaml 表记录某个用户的文件密钥，每个密钥都有 id，通过 id 来获取文件密钥。这样做的目的是为了以后用户可以自己指定文件密钥做准备。

至于用户密钥，这是最重要的一个，也是最需要防范的一个，所以，如图 6.18 对他做了比较复杂的逻辑。


```
class UserKeyNode{
public:
    int userId{};
    std::string userName{};
    std::string userPasswdHash{};

    int tableIndex[USER_KEY_INDEX_ROW][USER_KEY_INDEX_LINE];
    unsigned char sortHave[USER_KEY_INDEX_ROW]{};
    unsigned int mask{};

public:
    UserKeyNode() = default;

    UserKeyNode(const YAML::Node &node) {
        (*this) << node;
    }

public:
    void operator<<(const YAML::Node &node);

    void operator>>(YAML::Node &node) const;

public:
    std::string getFinalUserKey() const;
};
```

这是一张 30*500 的随机字符表,用户密钥的部分数据就是从它里面得到的。

在获取用户密钥的信息中存储以下信息: 一个 30*2 的二维数组, 指示使用密钥表中的那些数据; 一个无符号的 char[30], 将它当 short 使用, 指示了上面数组所指向的数据的排序权重。

然后按照上面的权重对所指向的表中的数据进行排序, 构成了初版的用户密钥。

在记录一个无符号整数掩码, 使用后 30 位, 将排序好的密钥, 主密钥分割为 30 份, 按这个掩码进行组合, 只有掩码中指示可以使用的部分才会被记录到用户密钥结果中, 其余部分将会被舍弃, 而使用主密钥对应的部分, 得到了第二版的用户密钥

最后将得到的数据按一定的规则进行转换的到用户密钥, 这个转换规则可以是 base64 编码, 也可以是 MD5 或 sha256 求 hash, 也可以直接使用, 这个可以 随时切换。

用户 UserKeyNode 通过一张 UserKeyNodeTable 进行存储, 最终通过 yaml 存储, 使用主密钥加密 yaml 表。至于具体地转换逻辑, 这里就不再贴代码进行讲述了, 上文类似的已经讲的比较清楚了。他会负责登录, 注册, 修改密码/用户名等操作, 这些都比较简单, 就不再细说。

对于文件密钥, 因为他会经常变动, 甚至用户还会加自己的用户密钥进来, 因此, 而我们又有一个比较安全的用户密钥, 所以直接使用一张表保存 id 和密钥字符串就好, 将序列化的 yaml 数据使用用户密钥加密存储。

当然他与用户表不同的是, 文件密钥是有可能被删除的, 所以他的 id 就不能使用简单的递增的方式了;

```

int FileKeyTable::insertKey(const std::string &key) {
    int _id{};
    do {
        _id = rand();
        auto dest = tab.find(_id);
        if (dest == tab.end()) {
            break;
        }
    } while (true);
    FileKeyNode node;
    node.id = _id;
    node.key = key;

    return insertKey(node);
}

```

图 6.20 随机不重复的文件密钥的过程

如图 6.20，通过随机一个文件密钥->检查是不是已经在文件密钥表中了->再插入的逻辑，就可以使用不会重复的文件密钥了。

6.6 src/gui_qt 利用 QT 库实现一个图形界面

6.6.1 gui 实现方式对比与分析

使用核心层提供的接口利用 QT 库实现一个图形界面，调用逻辑层服务。

原来打算将 qt 的主界面也实现为一个动态链接库，同时交给 main_controller 管理，main controller 通过不同的编译选项或者命令行参数，决定是通过命令执行一些逻辑，还是启动 qt 图形界面。

但是这里有一个很严重的问题，Cygwin 中的 qt5，如图 6.21 必须在启动一个 x11 提供的舞台之后，才能在这个舞台上展示他的窗口，这就变得非常丑。

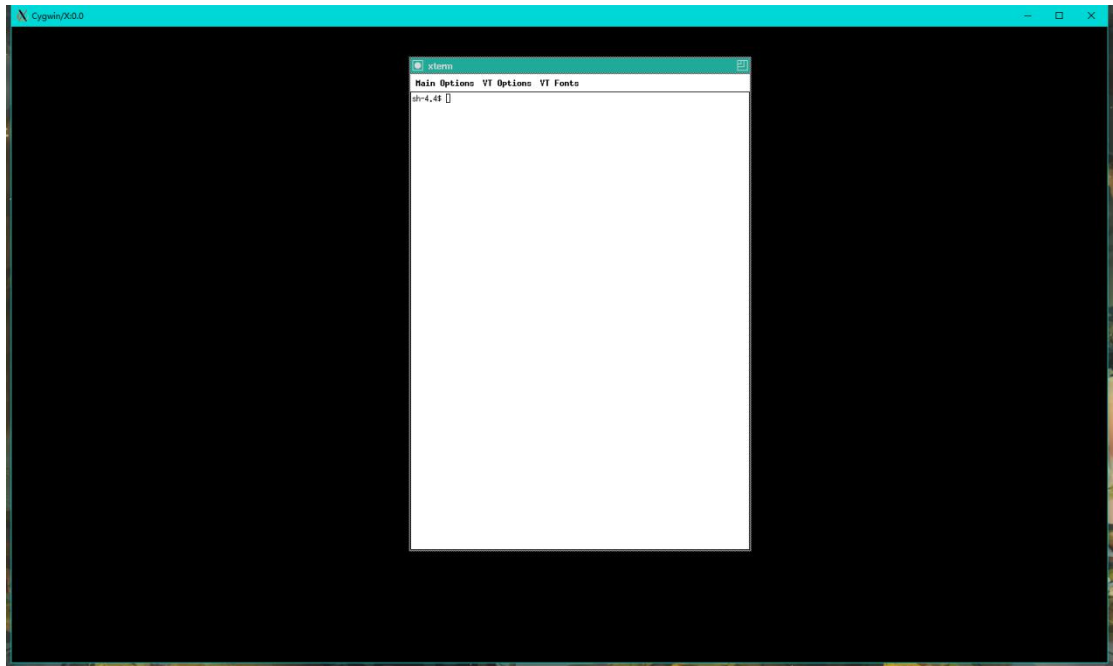


图 6.21 x11 提供的舞台

就是在这个舞台上在提供窗口服务，原因是 Windows 不支持 x 协议。但是，这样的效果太差了，因此只能舍弃这种方式。

由于 cygwin 对于 qt5 支持不好，反而，qt 是默认支持 Windows 的，所以又考虑不用费劲的使用 cygwin 在绕一圈，直接 include main_controller 和 web_adaptor 两个模块的头文件，然后在 qt 中通过动态链接库的方式加载已经编译好的动态链接库，这样，gui 层就绕过了 Cygwin。但是这样还是存在问题。虽然Cygwin生成的也是dll库，可以在Windows下直接运行，但是他们和Windows下直接使用 vs 或者 mingw 生成的 dll 库格式还是有所不同的，不能混合调用，这种方式也行不通。

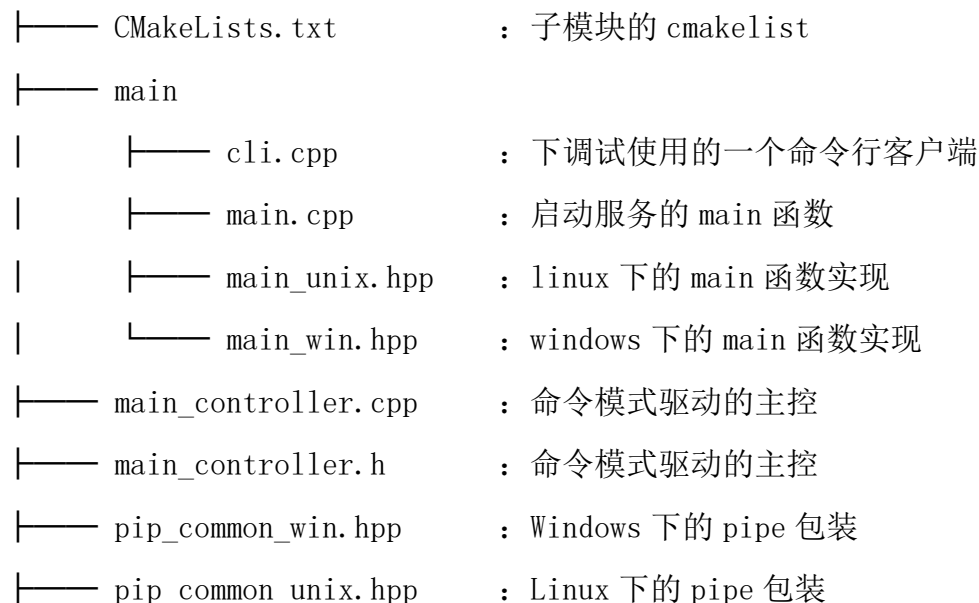
最终我选择了一种万金油的方式：远程访问+命令模式驱动。使用管道（安全）或者 socket（不安全）传输客户端的请求，通过命令模式驱动文件加密服务运作，这样，文件加密系统就被当作了一种服务，客户端这是一个命令生成器而已。这样即使客户端崩溃了，用户不小心点击退出了，也不会造成文件加密/解密被中断造成各种意外错误，十分稳定，当然通过管道传输数据肯定不如直接内存访问速度快，但是考虑到个人文件加密系统中，只是传输命令，这都是一些非常短的字符串，最长的也就是返回的某个文件夹下的文件名表，这也不会非常长，完全可以做到无感。

Gui 也不一定要使用 qt，通过这种方式，可以使用任意支持 pipe 的语言，

如：通过 jni 技术使用 javafx, mfc, python 的 pyqt 等等，gui 层完全被隔离出核心逻辑了。

6.6.2 命令模式主控的实现

首先要实现主控，塔方负责各种资源的调度，命令的解释，驱动整个系统运作，至于他的客户端是什么，完全不需要它关心，他只负责掌管各种适配器，和根据命令对适配器进行操作。



如图 6.22，在主控中使用一个 enum 定义各种命令，通过一个以 string 为 key, int 为 value 的 map 让 c++ 的 switch 具有字符串作为 key 的功能。

```
enum OPERATIONS {  
    NONE = 0,  
    REGISTER,  
    LOGIN,  
    LS,  
    MV,  
    CP,  
    RM,  
    ADD,  
    DEC,  
    ADD_KEY ,  
    DEL_KEY = 10,  
    SYNC,  
    LS_KEY,  
    START_WEB,  
    STOP_WEB,  
    STATUS_WEB,  
    QUIT,  
};
```

图 6.21 命令的 enum 定义

这就是所有的命令了，通过命令和 3 个额外字符串参数，可以实现常用的所有文件操作的命令。

```

switch ((*STR_TO_ENUM)[cmd]) {
    case REGISTER://done
        if (userKeyTable->loadKeyTab()) {
            ERROR_LOG("init user key table faild!")
            iRet = -1002;
            break;
        }
        iRet = userKeyTable->registor(para1, para2);
        userKeyTable->savrKeyTab();
        break;
    case LOGIN://done
        iRet = initAndLogin(para1, para2);
        break;
    case LS://done
        nodes.clear();
        fileController->selectFileWithDirByName(para1, &nodes);
        iRet = 0;
        rsp.data=MainFileController::writeList(&nodes);
        if(rsp.data.empty()) {
            rsp.data="none";
        }
        break;
    case HV://done
        origin = nullptr;
        origin = fileController->selectByName(para1);
        if (origin == nullptr) {
            iRet=-1003;
            ERROR_LOG("no file named " << para1)

```

图 6.23 在 switch 中调用包装好的各种方法

最终，如图 6.23 在 switch 中调用包装好的各种方法，驱动整个文件加密系统运作。

至于访问的协议，如图 6.24，采用 req 对象->yaml 字符串->加密的数据->base64 编码的数据的流程进行访问。

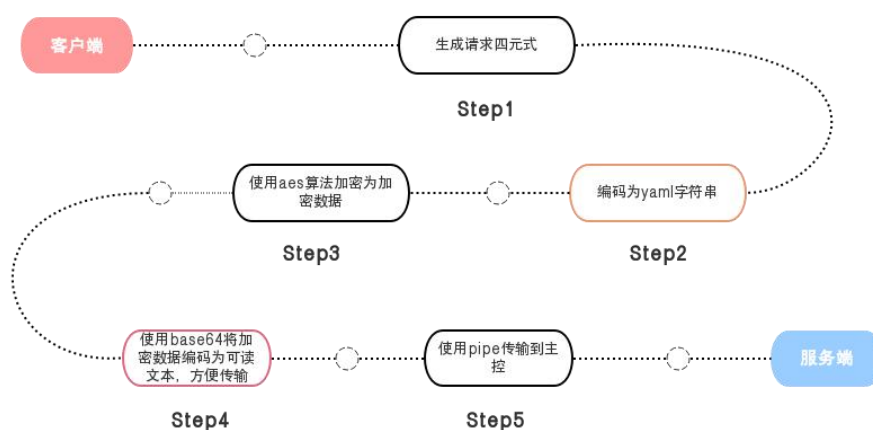


图 6.24 客户端请求流程

如图 6.25, 当请求到达主控后, 主控解包请求, 将他转换为 c++对象, 并通过命令模式驱动主控进行相关操作, 然后回包。

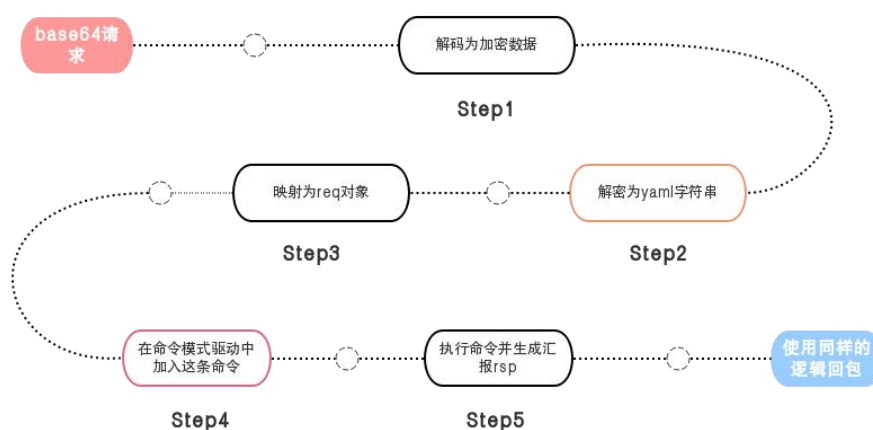


图 6.25 主控响应流程

6.6.3 命令行客户端的实现

这样, 我们就可以使用任意 gui 库生成我们的 gui 了, 也可以非常简单地使用 cli 生成命令作为客户端驱动:

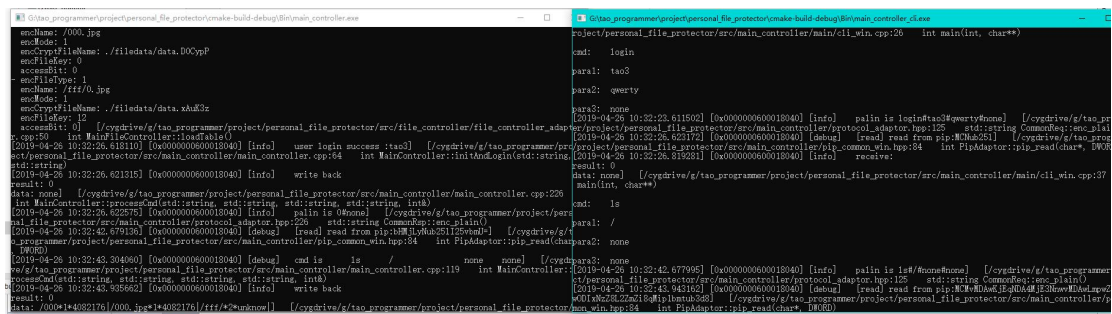


图 6.26 一个命令行写的简单客户端，用于测试

如图 6.26，这是一个 cli 驱动的客户，当然只是作为测试使用的，它输入测试数据要比 gui 方便很多。

6.6.4 qt 客户端的实现

Qt 的 qml 是 c++ 跨平台 gui 实现的一种简便方式，写起来很想 css+JavaScript，性能也很不错。QML 是一种描述语言，并且它们已经被整合到 Qt 开发环境中，QtQuick 的核心之一就是 QML 语言。QML 元素是先进的图形元素，这些图形元素可以叠加在任意图形界面上，像搭积木那样构建程序界面，这一特性使得 QML 既可以快速创建新的程序界面，也可以非常方便地改造已有的程序界面，QML 还可以在脚本里创建图形对象，并且支持各种图形特效，同时又能跟 Qt 写的 C++代码进行交互，使用起来非常方便。^[12]如图 6.27，就是利用 qml 实现的一个简单界面的展示。

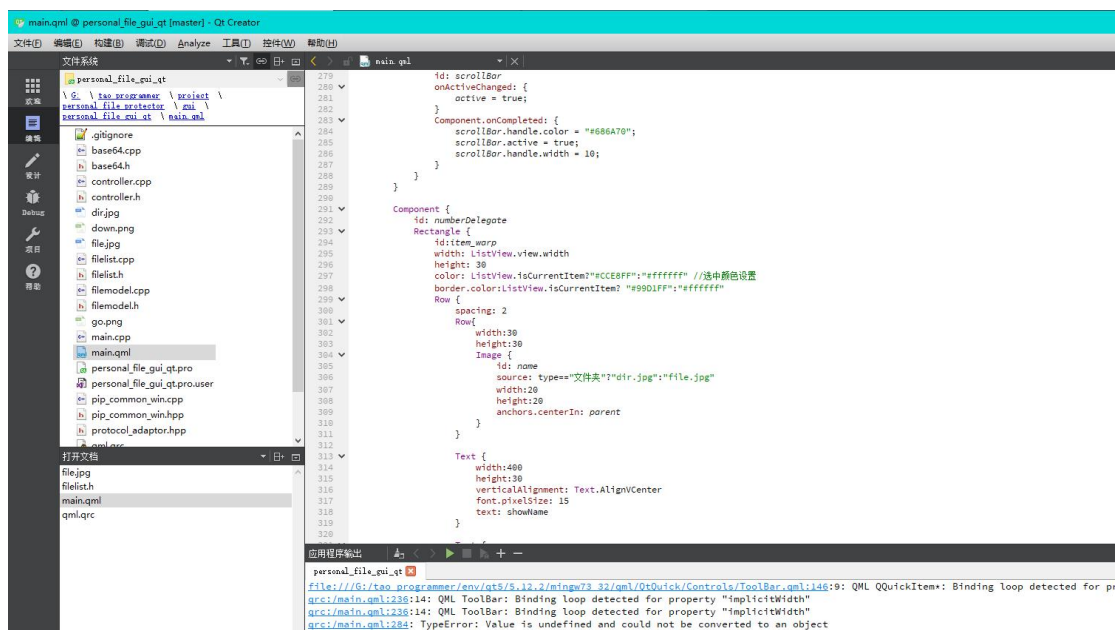


图 6.27 利用 qml 实现的一个简单界面

图 6.28 6.29 6.30 时对结果的简单展示。

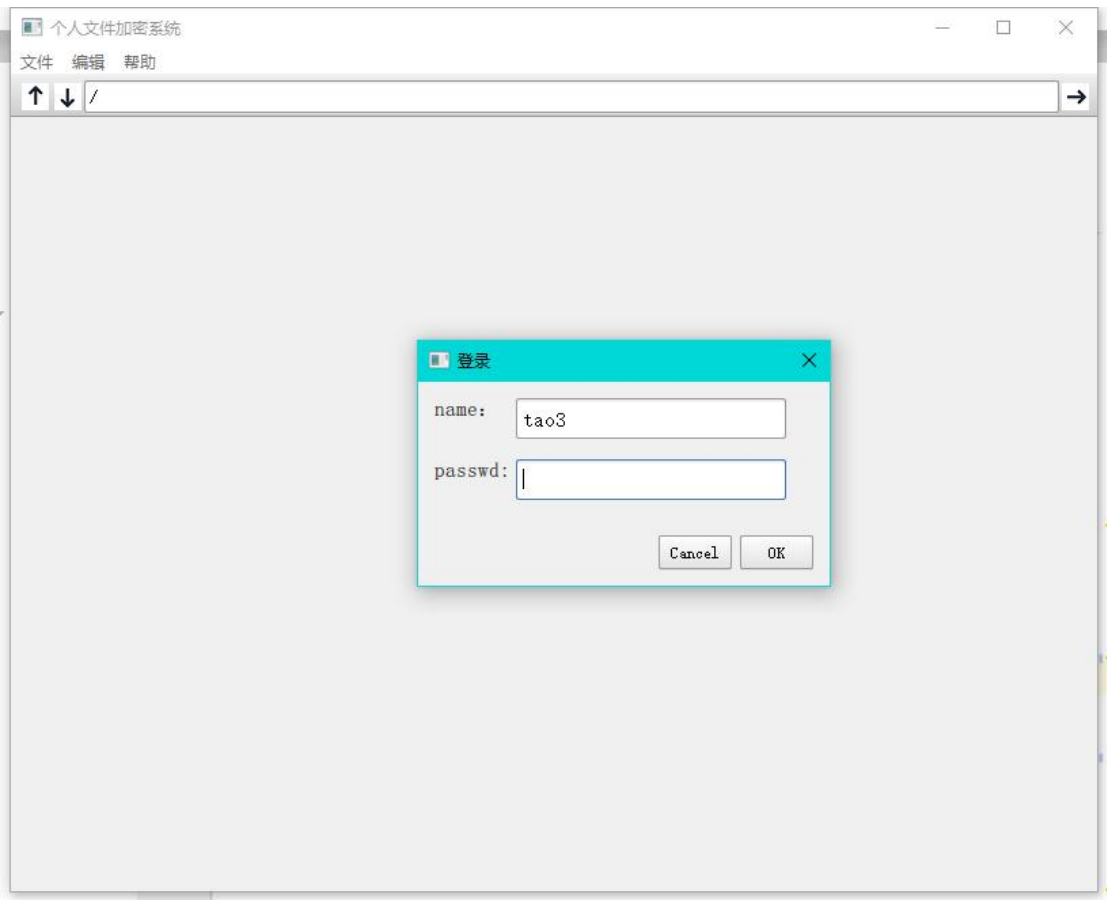


图 6.28 结果展示

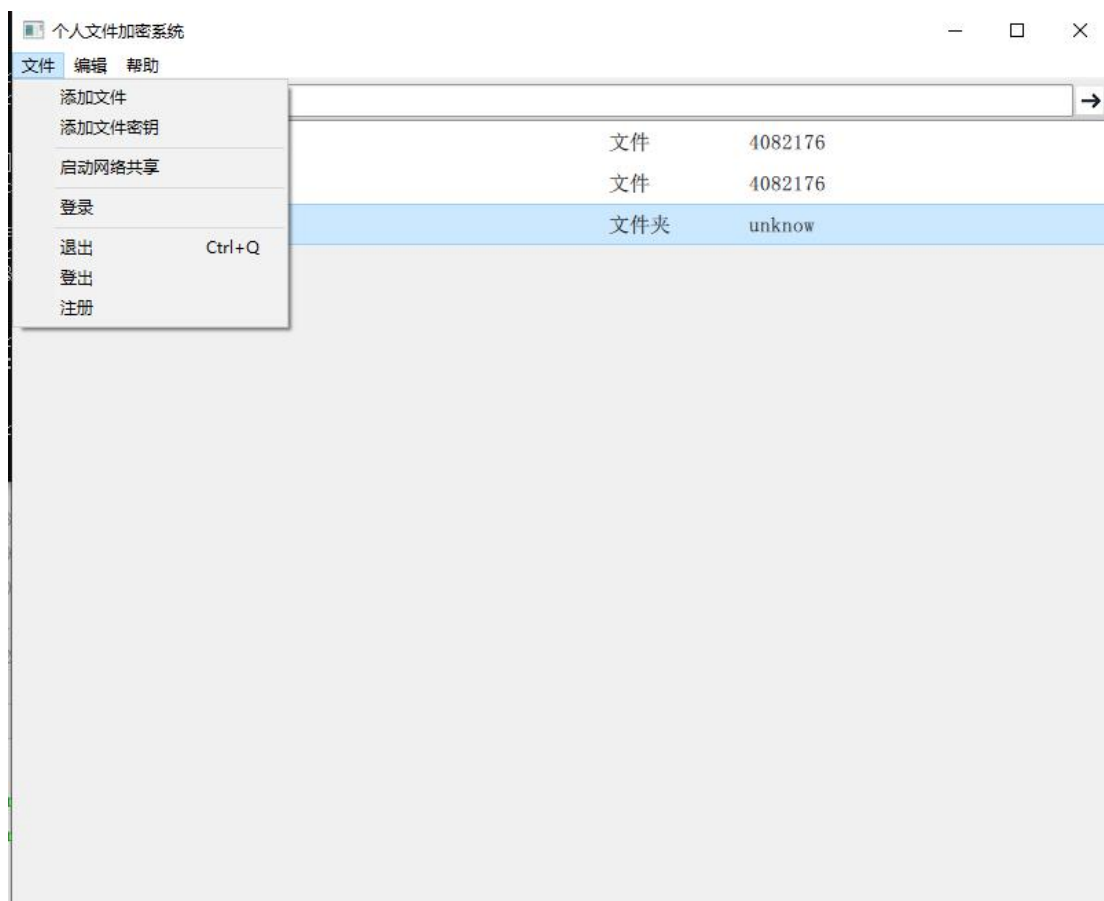


图 6.29 结果展示

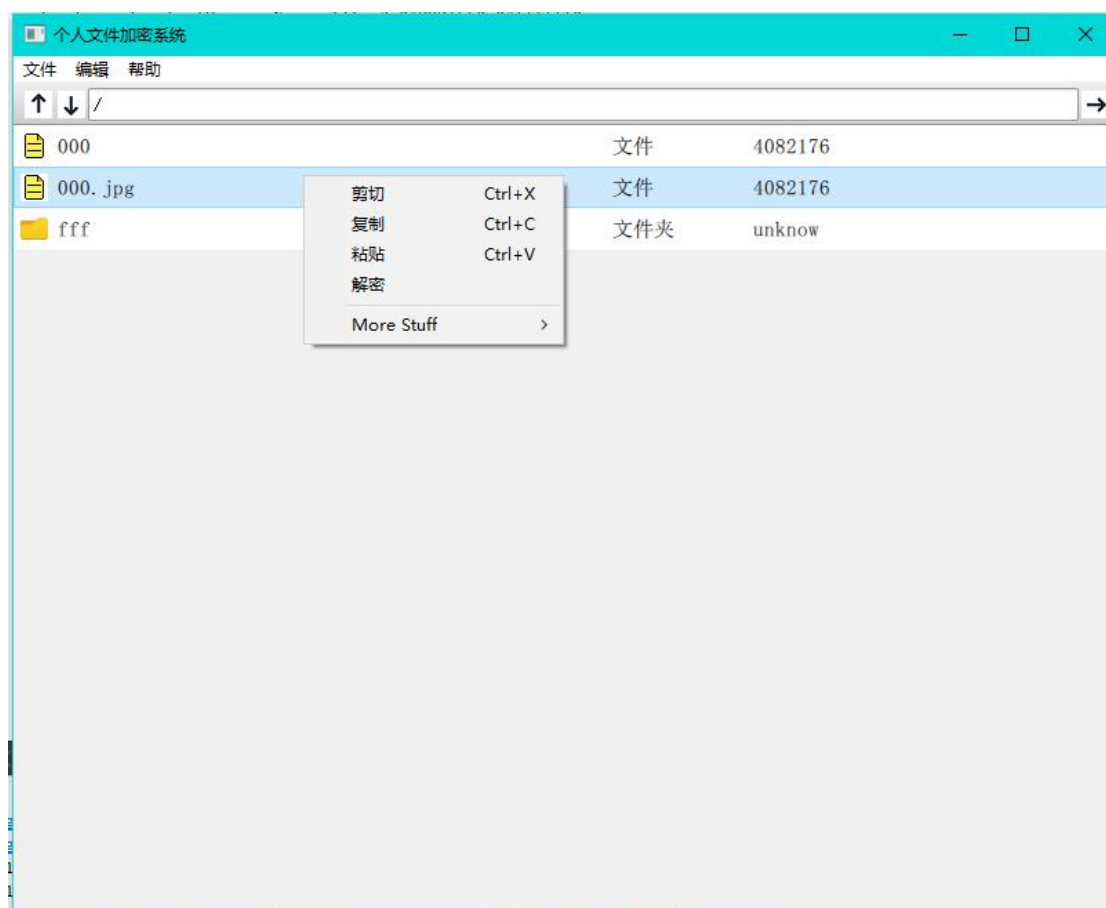


图 6.30 结果展示

这里完全仿照 windows 资源管理器的操作习惯,除了双击时解密到指定文件夹和拖入文件按,拖出文件会慢一些外,其他的和 Windows 的资源管理器完全兼容。当然这里只是对功能的实现,界面的美化我并不擅长,所以只能先做到这样了。

6.7 src/web_adaptor: 简单的网络实时解密浏览实现

```
class WebController {
public:
    static WebController *Instance;
public:
    MainFileController *mainController;
    int status 0;
public:
    WebController() = default;

public:
    std::string GetDirHtml(std::string name);

    int TranshFile(std::string name, size_t currSize);

public:
    int startServer(int port = 2333, std::string host = "localhost");

    int stopServer();

    int getStatus() {
        return status;
    }
};
```

图 6.31 网络访问接口

如图 6.31 时网络访问接口，通过它的拉起的 main_controller 将一个文件主控的地址给他，通过组件模式让他也拥有文件主控的功能。

由文件适配器提供获取文件夹下内容的方法和将文件解密到 fd 的方法，他们只是为了逻辑独立，使用适配器模式对文件主控的一个包装，方便以后更换文件主控，或者文件主控修改接口时，减少要修改的地方。

至于文件服务器，则是一个非常简单的 html 服务，访问者每次都带回来一个目标文件，如果是文件夹，则调用 GetDirHtml，返回新的文件列表的 html，如果是文件，那么就先写一个文件 html 响应头，然后将文件解密到 socket 的文件描述符中。

这样用户就可以通过浏览器不用解密就可以访问文件了，而且还支持通过 WiFi 进行手机访问，如果路由器允许，甚至可以实现外网访问。

但是对于多用户同时访问的实现，则走了一些弯路：刚开始时，如图 6.32，采用 unix 下常用的 select/poll 模型进行开发：

```
//new conn comes
do {
    if (FD_ISSET(svr_fd, &curr_set)) {
        client_fd = accept(svr_fd, (sockaddr *)&client_addr, &client_addr_len);
        if (client_fd < 0) {
            ERROR("something wrong happened when accept" << strerror(errno))
            break;
        }
        for (i = 0; i < FD_SETSIZE; i++) {
            if (client_fds[i] < 0) {
                client_fds[i] = client_fd;
                i++;
                break;
            }
        }
        snprintf(svr_buff, sizeof(svr_buff), "%s:%d",
            inet_ntop(AF_INET6, &client_addr.sin6_addr.__in6_u, svr_buff1,
                sizeof(svr_buff1)), ntohs(client_addr.sin6_port));

        if (i >= FD_SETSIZE) {
            ERROR("too many conns, throw this :") << svr_buff
            break;
        }

        INFO("conn from " << svr_buff)

        FD_SET(client_fd, &all_set);

        if (i > max_client_i) {
            max_client_i = i;
        }
        if (client_fd > max_fd) {
            max_fd = client_fd;
        }
    }
} while (false);

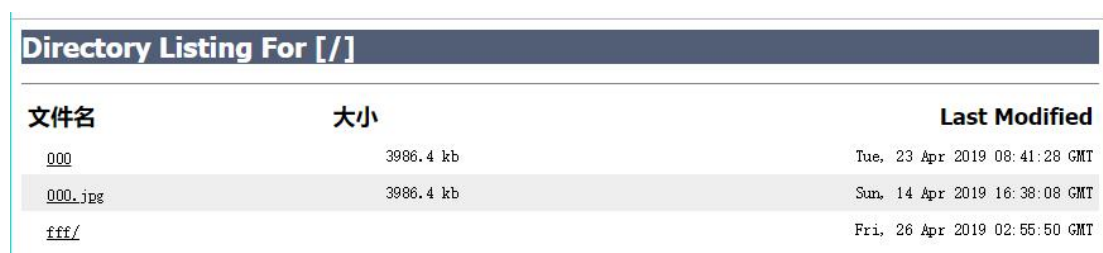
if (FD_ISSET(svr_fd, &curr_set) && (--ready_number <= 0)) {
    continue;
}
```

图 6.32 select/poll 模型

但是，在实际体验时，发现 select 当一个用户在在线使用文件时，其他用户会卡死，思考了一下发下犯了一个低级错误：select 函数用于确定一个或多个套接口的状态，对每一个套接口，调用者可查询它的可读性、可写性及错误状态信息，用 fd_set 结构来表示一组等待检查的套接口，在调用返回时，这个结构存有满足一定条件的套接口组的子集，并且 select 返回满足条件的套接口的数目。^[13]select 本质上还是只有一个线程在工作，当这个线程上的用户调用解密函数时，这个函数不会退出，要花费很长时间才能完成，这段时间内，其他用户就无法得到响应了。

因此，与其在解密函数处调用线程分离主线程，还不如直接使用多线程版的网络模型，简单又不会卡住。Select/poll 模型，适用于同时又很多链接同时打开，但同一时间只有很少的链接在活跃的场景，在这种场景下，建立大量的连接，

如果使用线程的话,那么线程切换将会消耗大量的资源。而我们的场景刚好相反,只有很少的连接建立,但是,一旦激活传输文件,那么这个连接将会使用很长时间,所以这种情况下,为每一个连接建立一个连接,通过锁控制并发是最好的选择。通过拼接 html 网页文本,就可以实现简单的网页浏览了。如图 6.33,就是展示界面。



Directory Listing For [/.]		
文件名	大小	Last Modified
000	3986.4 kb	Tue, 23 Apr 2019 08:41:28 GMT
000.jpg	3986.4 kb	Sun, 14 Apr 2019 16:38:08 GMT
fff/		Fri, 26 Apr 2019 02:55:50 GMT

图 6.33 结果展示

但是由于现在还没有对网络权限进行验证,依赖客户端的验证,这个模块还需要完善。基于上述理由,网络模块的命令中就只开放了 ls 命令,不允许网络客户端对文件进行操作。

第七章 不足与期待

7.1 实现一个 NFS 服务器

mmap 一段内存,将文件数据也映射到这里来,进行读写访问(整个过程全在内存中,依赖内存的交换机制),最后对整块内存进行加密,写回。可以使用 windows 自带的 mount 技术,将 NFS 挂载到一个盘符下,直接使用 windows 资源管理器操作(复杂)。

NFS 客户端和 NFS 服务端通讯过程:首先服务器端启动 RPC 服务,并开启 111 端口;服务器端启动 NFS 服务,并向 RPC 注册端口信息;客户端启动 RPC 向服务端的 RPC 服务请求服务端的 NFS 端口;服务端的 RPC 服务反馈 NFS 端口信息给客户端;客户端通过获取的 NFS 端口来建立和服务端的 NFS 连接并进行数据的传输。

7.2 脱离 cygwin 而实现全平台编译

在之前的设计中,之所以使用 cygwin 而非 windows 自带的 vs 进行编译,是因为考虑到在不同的平台下,对文件的细节操作完全不同,所以使用 cygwin 进行

统一，而且 Windows 对 `select / poll` 的支持非常弱，以前的设计中是会用到 `select / poll` 模型的，基于这些考量，最终选择了 Cygwin 作为 Windows 下的编译器。

但是使用 Cygwin 也不是没有代价的，在 Cygwin 中很多类库是没有官方版本的二进制包的，必须得自己下载源码进行编译。这是非常慢的。

而且，有一些类库，使用了汇编进行加速，那么这是后的编译就要非常小心，`openssl` 库在编译时，又一些汇编实现的就无法指定平台，只能退而求其次，使用费汇编版本的通过编译，而最早使用的 `log4cplus` 日志库，就是由于最新版本对 Cygwin 有兼容问题，只能换掉，换成了 `boost::log`。

因此在实现中，我尽量避免使用和操作系统直接相关的函数，在设计中，如果这个量足够小，而且有可跨平台替代版本的话，就可以考虑取消 Cygwin。

现在，在所有功能基本实现的情况下，回头统计了一下，发现和底层相关的调用，有：

`mmap` 函数，win 版和 unix 版是不一样的，但可以考虑通过编译宏来控制。

`mkstemp`，windows 下暂时没找到，但是第三方库中又提供，或者自己写一个也行，只是随机一个不重复的文件名，在已经有所有资源的情况下并不难。

网络访问，这一块 Windows 和 unix 很多不一样，但是 `boost` 库有对一部网络 `io` 进行跨平台的封装，可以考虑使用 `boost::asio` 库。

至于其他的操作，文件操作统一使用 `boost::filesystem` 库，加密解密使用 `openssl crypt` 库，`yaml` 使用 `yaml-cpp` 库，日志使用 `boost::log` 库这些都是跨平台支持的，因此理论上只要把上面的三个问题解决一下就可以抛开 Cygwin 而实现真真的跨平台了。

7.3 对网路访问进行权限验证

现在是没有网络验证的，只是通过路径进行了用户的区分，但是当这个用户打开网络共享之后，所有可以访问到用户网络的人（如连接了该用户 WiFi 的人）都可以访问到该用户的文件。

要实现登录鉴权，本身有大量的例子可以参考，但是这又是一个非常庞大的功能，涉及到 `session` 的保存，过期，验证，等复杂的逻辑，因此，有两种解决

方案：

7.3.1 自己实现简单的网络链接管理

HTTP 是无状态的，就是前后两个 HTTP 事务它们并不知道对方的信息。为了维护会话信息或用户信息，一般用 Cookie 或 Session 技术缓存信息。Cookie 是存储在客户端的，Session 是存储在服务端的。

客户端请求服务器时，如果请求的服务涉及 Session 的访问，如果请求中包含 session id，则从 session 管理器中获取 session，如果不包含 session id，则会生成一个新的不冲突的 session id。一般是通过 Cookie 随 Request 和 Response 在客户端和服务端间传输。在 Cookie 禁用情况下，也可由 URL 参数的形式进行通讯。^[14]

这种方式逻辑全部集中在自己的代码中，方便发布，没有外部依赖。

7.3.2 使用 apache 服务器的 cgi 功能

继承一个 apache 服务器进去，通过 cgi 的形式实现网络访问，而非自己再去写网络访问逻辑。

Apache 服务器本身是有 session 管理的，省去了自己实现，减少 bug 的可能。

但是这回导致项目中必须包含一个 apache 服务器出去，比较臃肿，而且 apache 所提供的大部分功能我都用不到。

这两种实现方式还得评估一下。

7.4 网络穿透

现在的网络访问是可以通过和用户在同一局域网下的用户访问，如该电脑的 WiFi 链接的手机，同一路由器下的其他用户等等，但是一旦离开了居于网，就无法访问了，这是因为现在用的 ipv4 地址不够用，因此我们很难拥有一个通用的地址，大部分时候都是网络管理员分配的局域网地址，在网络出口哪里进行转化，这使得我们要在外网访问我们内网的服务器，会很麻烦，有以下两种解决方案：

7.4.1 NET 转换

在 NAT 网关（如：路由器，局域网出口）上会有一张映射表，表上记录了内

网向公网哪个 IP 和端口发起了请求，然后如果内网有主机向公网设备发起了请求，内网主机的请求数据包传输到了 NAT 网关上，那么 NAT 网关会修改该数据包的源 IP 地址和源端口为 NAT 网关自身的 IP 地址和任意一个不冲突的自身未使用的端口，并且把这个修改记录到那张映射表上。最后把修改之后的数据包发送到请求的目标主机，等目标主机发回了响应包之后，再根据响应包里面的目的 IP 地址和目的端口去映射表里面找到该转发给哪个内网主机。这样就实现了内网主机在没有公网 IP 的情况下，通过 NAT 技术借助路由器唯一的一个公网 IP 来访问公网设备。^[15]但这需要用户有权限修改网关或者让网关管理员修改，而且 ISP 分配给普通用户的 ip 地址通常不是固定的，这又需要我们经常更改链接地址。

7.4.2 反向代理

通过在公共端点和本地运行的 Web 服务器之间建立一个安全的通道，实现内网主机的服务可以暴露给外网。可以考虑把这个功能直接集成到一个工具中，在搭建一个反向代理工具，让有需要的用户拉起反向代理，而没需要的就不用理，杜绝加密管理器联公网的可能（这个加密管理器的设计基本原则之一）。

7.5 文件压缩

可以考虑使用压缩技术将加密后的文件进行压缩，这样可以节省磁盘空间。通用的 7z 提供的 LZMA SDK 就很不做，提供较好的 c++ 支持。当然 openssl 本身也提供压缩支持：

6.1 LZMA SDK includes:

C++ source code of LZMA Encoder and Decoder

C++ source code for .7z compression and decompression (reduced version)

ANSI-C compatible source code for LZMA / LZMA2 / XZ compression and decompression

ANSI-C compatible source code for 7z decompression with example

C# source code for LZMA compression and decompression

Java source code for LZMA compression and decompression

lzma.exe for .lzma compression and decompression

7zr.exe to work with 7z archives (reduced version of 7z.exe from 7-Zip)
SFX modules to create self-extracting packages and installers
ANSI-C and C++ source code in LZMA SDK is subset of source code of
7-Zip.

LZMA features:

Compression speed: 3 MB/s on 3 GHz dual-core CPU.

Decompression speed:

20-50 MB/s on modern 3 GHz CPU (Intel, AMD, ARM).

5-15 MB/s on simple 1 GHz RISC CPU (ARM, MIPS, PowerPC).

Small memory requirements for decompression: 8-32 KB + DictionarySize

Small code size for decompression: 2-8 KB (depending on speed
optimizations)

The LZMA decoder uses only CPU integer instructions and can be
implemented for any modern 32-bit CPU.^[16]

根据 7z 的官方文档来看是很符合我们的需求的。在加密文件时，先把加密文件写到一个 mmap 的不存储到本地问文件映射中，然后同时在另一个线程读取这个文件，并进行压缩，输出到目标文件中，这是要注意一定要把已经压缩的部分 ummap 掉，不然处理大文件时内存会被占满，操作系统很难判断到底哪些该换出。

这个可控性比较高，7z 的速度，压缩率都是值得信赖的。

Openssl 的 COMP_zlib

COMP_zlib 会返回基于 zlib 库的 COMP_METHOD，和加密的逻辑保持一致，只需要要加密文件过滤器之后再加一个压缩过滤器就好，实现起来非常方便，由于时间原因，这里没有再进行包装实现。

第八章 结语

这个文件个人文件加密管理器就基本实现了，提供了三级密钥管理，主密钥写死在动态链接库中，通过直接修改动态链接库来为每个用户提供不同的主密钥；用户密钥通过查表加排序，并和主密钥用掩码进行互补保持，使用主密钥加

密的 yaml 文件按存储相关信息；文件密钥使用用户密钥加密；文件使用文件密钥进行加密；同时提供了 gui 访问和通过局域网网络不需要解密就进行访问，并且支持手机访问；当然除了主要功能之外，还有一些额外功能没有来得及实现，还需要继续优化。

参考文献

- [1] W.Stallings & L.Brown, (2012) “Computer Security, Principles and Practice”, 2nd/e, Prentice Hall.
- [2] 张华. RSA 算法对信息保护的意义和作用[J]. 科技风, 2018(31):78-79+87.
- [3] 李翔宇, 于景泽. DES 加密算法在保护文件传输中数据安全的应用[J]. 信息技术与信息化, 2019(03):23-25.
- [4] 李青, 陈靓, 冯梅, 李程辉, 方静. 浅析几种典型数据加密算法[J]. 信息系统工程, 2017(11):148-149.
- [5] 魏革. IDEA 加密解密算法的设计与实现策略探究[J]. 无线互联科技, 2015(24):58-59.
- [6] 詹鹏伟, 谢小姣. DES 与 AES 算法实现及其在图像加密中的效率探究[J]. 网络安全技术与应用, 2018(09):41-42.
- [7] <https://www.cnblogs.com/block2016/p/5596693.html> 分组密码的工作模式 作者: block2016
- [8] 裴宏韬, 孙建言, 盖红玉, 李稳. 对称加密算法分析及用 Java 实现[J]. 电脑知识与技术, 2017, 13(18):46-47+61.
- [9] 张浩然, 曾文潇, 蒋同海. 用 Java 和 OpenSSL 实现认证中心[J]. 计算机应用研究, 2004(05):157-159.
- [10] <https://blog.csdn.net/liao20081228/article/details/77193729> OpenSSL 中文手册之 BIO 库详解 作者: 蓝月心语
- [11] <https://baike.baidu.com/item/mmap/1322217> 百度百科 mmap 函数
- [12] 冯源. QML 语言在显控界面开发中的应用[J]. 电脑编程技巧与维护, 2018(02):62-64.
- [13] <https://baike.baidu.com/item/select%28%29/10082180?fr=aladdin> 百

度百科, select 函数

[14] <https://www.cnblogs.com/nick-huang/p/6660232.html> Tomcat 中利用 Session 识别用户的基本原理 作者: Nick Huang

[15] <https://www.jianshu.com/p/cdc446e51675> 可以实现内网穿透的几款工具 作者: 哥本哈根的小哥

[16] <https://www.7-zip.org/sdk.html> 7z document

致谢

很庆幸自己能够顺序里完成此次毕业论文,在这里我要对我的帮助过我的老师同学表示衷心的感谢。

首先需要感谢我的指导老师,徐武平老师,在我不知道如何去开始毕业论文的选题的时候,他悉心指导,帮助我在众多论文信息中理清思路,确定了选题。在毕业设计完成的过程中,徐老师不断关注我们的项目进展,定期让我们进行汇报,根据我们研究的结果提出建设性的意见,让我们将毕业设计内容完成得更加充实而有说服力。

其次我还要感谢学院,感谢对我言传身教地老师们,是您们的教学和训练,让我拥有了扎实的专业基础和初步科研的能力,这样才能顺利的完成找论文,读论文,提出课题,研究并得出结果的这样一个过程。

最后感谢武汉大学这样一个优秀环境,让我与很多优秀的同学交流学习,开阔了我的视野,增长了见识,同时也感谢学校提供的丰富的资料书籍期刊的使用权,让我可以无障碍地学习。感谢大学四年的时光让我成为了更好的自己。