**《安全编程》作品报告**

题目： MQTT协议逆向及模拟

班级： 网络空间安全2022级

姓名： 王际翔

教师： 张恒汝

2024年12月17日

[1 需求分析 1](#_Toc22014)

[1.1项目背景 1](#_Toc26001)

[1.2 功能分析 1](#_Toc17060)

[2 总体设计 1](#_Toc1654)

[2.1 体系结构设计 1](#_Toc28308)

[2.2 函数设计 1](#_Toc3439)

[3 详细设计 1](#_Toc5486)

[3.1 解压流程 2](#_Toc20779)

[3.2 解压 4](#_Toc24179)

[3.3 解压 6](#_Toc527)

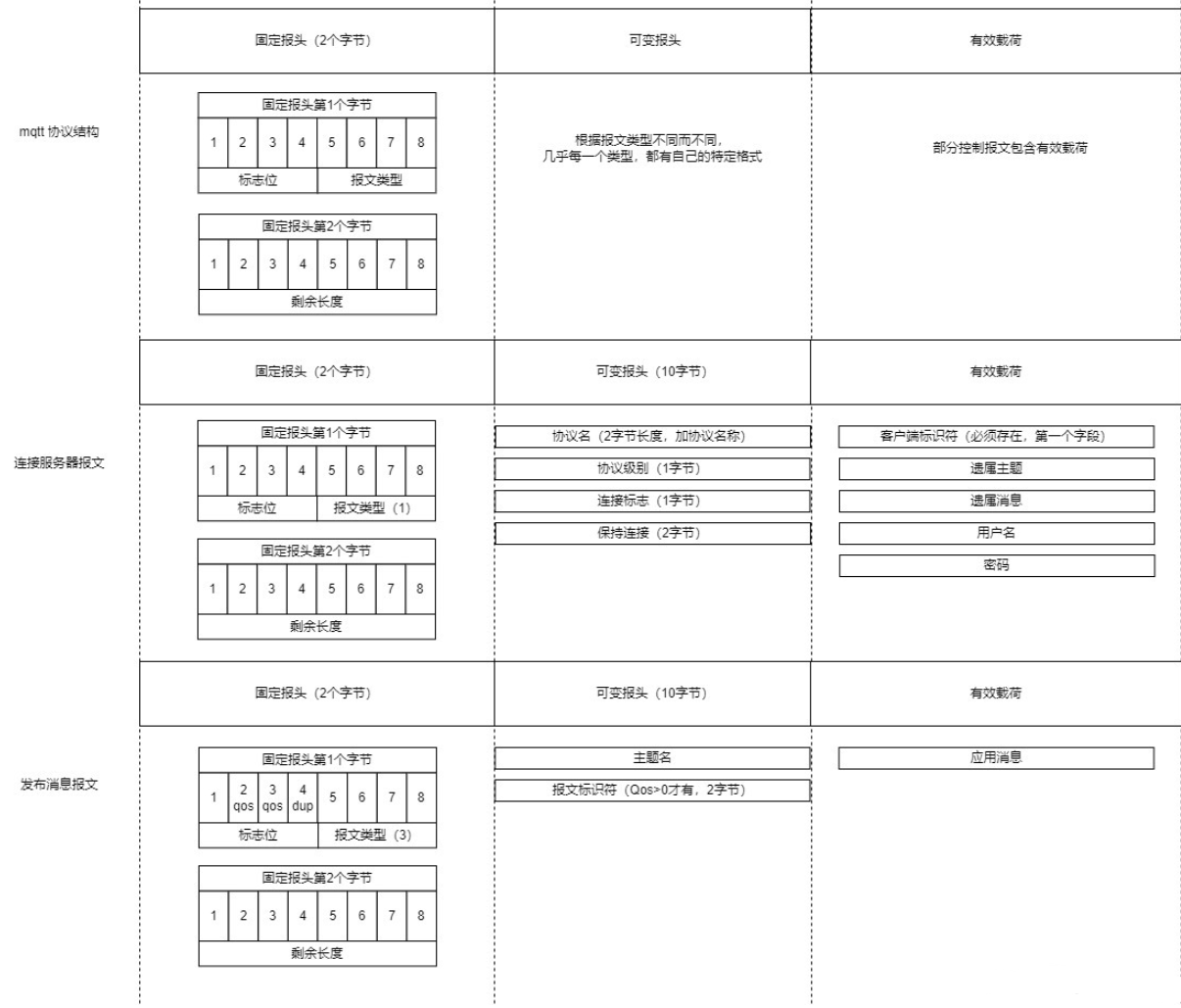
[4 总结 7](#_Toc19983)

# 1 方案与实现步骤

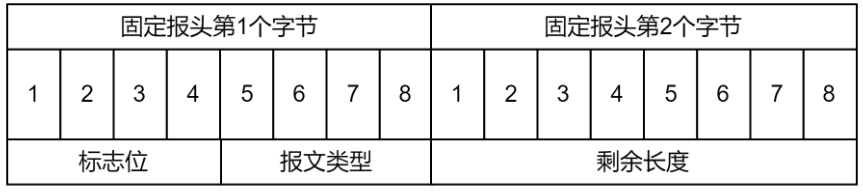
## 1.1MQTT协议提前了解

MQTT协议在IoT领域是使用的最广泛的通用协议。MQTT是一种基于发布/订阅（publish/subscribe）模式的轻量级通讯协议，该协议构建于TCP/IP协议上，由IBM在1999年发布。MQTT在官方解释中的优点是：用极少的代码和有限的带宽，为连接远程设备提供实时可靠的消息服务；简单理解就是，实现起来简单，并且在传输上无效的数据也很少，并且能够保证数据传输的可靠性。

MQTT协议由三部分组成：固定报头、可变报头、有效载荷。固定报头是所有报文的统一格式；可变报头则根据固定报头中的报文类型不同而改变，每个报文类型基本上都有自己的可变报头格式；有效载荷只是部分报文有。报文内容是根据报文类型的不同而不同。



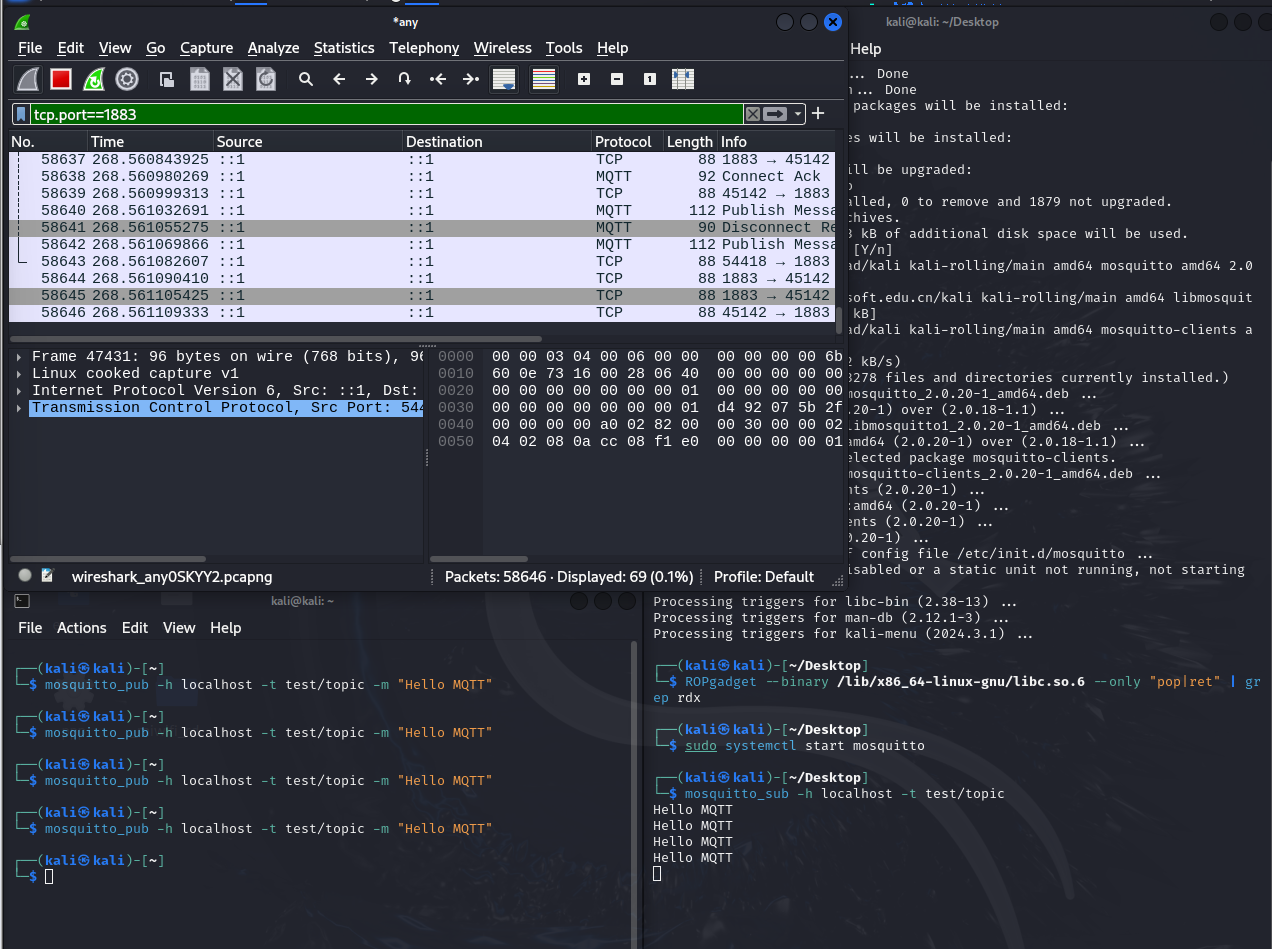
固定报头：一共两个字节

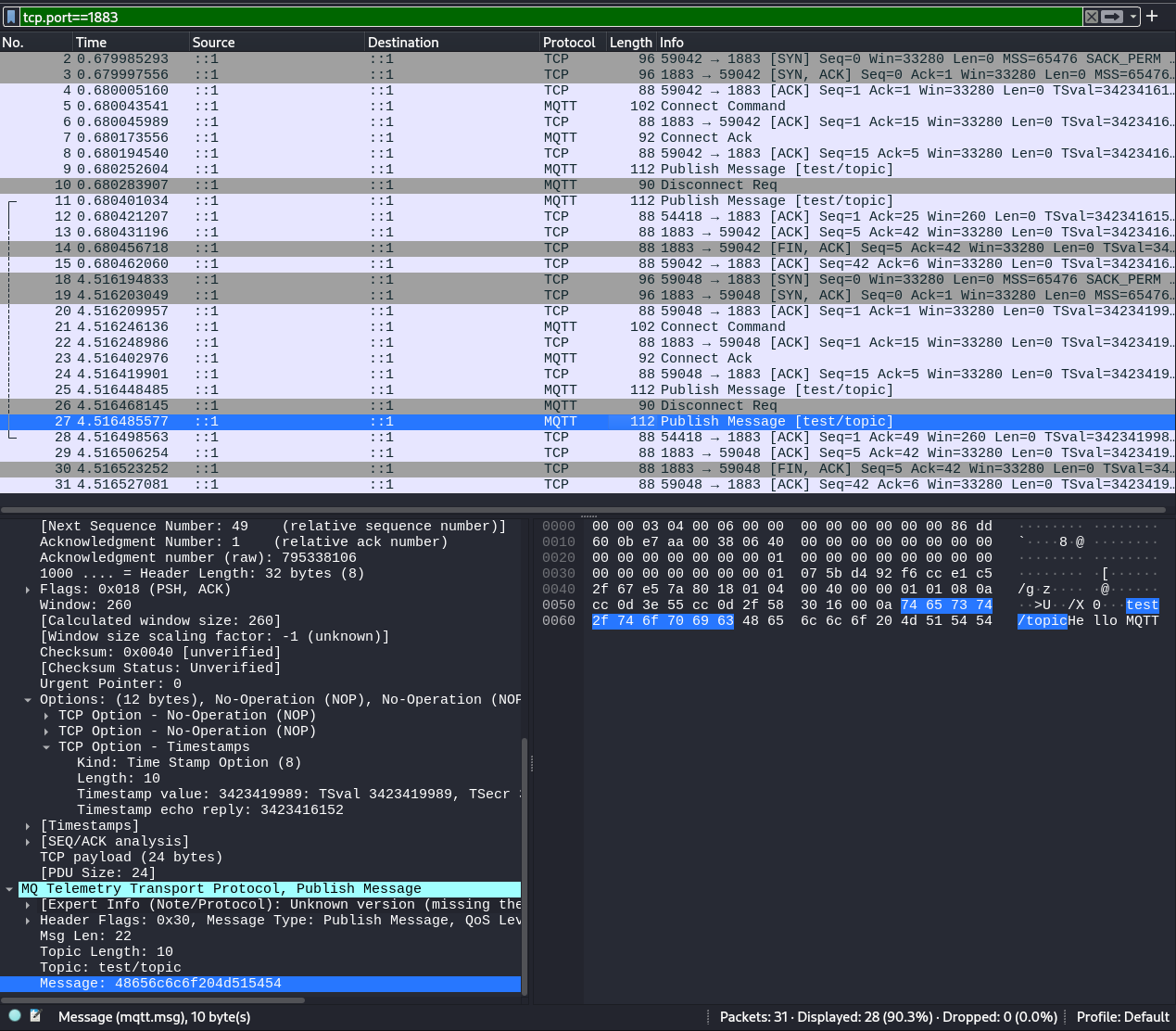


## 1.2捕获与分析MQTT数据包

### 1.2.1数据包捕获

首先安装对应包体，通过订阅对应ip的对应服务，再在其他地方向对应ip的对应服务发布消息，便会在订阅端接收到对应的消息。MQTT默认端口为1883，便可以使用wireshark监听对应端口的数据包。





前面提到MQTT是构建于TCP/IP协议上的，前面0x58(88)个字符并非MQTT数据包部分，接下来才是MQTT数据包部分。这里的数据包固定头部为0x30 0x16，其中剩余长度为0x16

## 1.2 功能分析

1. 多格式支持：支持常见压缩格式zip、tar、tar.gz、tar.xz、xz、gz、7z、tar.7z。
2. 进度显示：在终端显示每个文件的解压进度条，实时反映进度。
3. 错误处理：能够识别不支持的文件格式并提供错误提示。
4. 并行处理：能够同时处理多个文件，提升处理效率。
5. 性能：解压速度快，进度条实时更新。
6. 可靠性：在解压过程中处理错误情况，如文件不存在或格式不支持。
7. 可维护性：代码结构清晰，便于后续功能扩展和维护。

# 2 总体设计

## 2.1 体系结构设计

本文档描述了一款用于处理多种压缩格式文件解压缩的程序架构。该程序支持多种压缩格式，包括 zip, tar, tar.gz, tar.xz, xz, gz, 7z, tar.7z。通过使用 C 语言和 libarchive 库，让该程序能够高效地解压缩文件。

## 2.2 函数设计

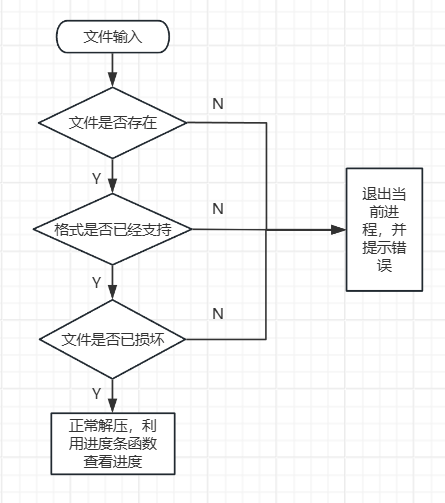
主函数目的是创建多个子进程进行一个或多个文件的解压；创建了一段所有进程都可读写的空间，方便实现进度条函数以及进度条更新；在多进程下，父进程会等待所有子进程完全退出才正常退出。

文件格式识别利用读取其二进制来实现，根据文件头部信息尝试识别文件格式，返回的字符串作为文件处理函数的参数尝试调用文件夹创建函数以及分别调用解压函数。

压缩文件通过archive库处理，更加便捷安全快速。

# 3 详细设计

### 3.1 解压流程



int main(int argc, char \*argv[]) {

if (argc < 2) {

fprintf(stderr, "用法: %s <压缩文件1> <压缩文件2> ...\n", argv[0]);

return EXIT\_FAILURE;

}

size\_t shm\_size = sizeof(int) \* argc;

int \*progress = mmap(NULL, shm\_size, PROT\_READ | PROT\_WRITE, MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

if (progress == MAP\_FAILED) {

//perror("mmap");

return EXIT\_FAILURE;

}

memset(progress, 0, shm\_size);

for (int i = 1; i < argc; ++i) {

pid\_t pid = fork();

if (pid < 0) {

//perror("fork");

return EXIT\_FAILURE;

} else if (pid == 0) {

process\_file(argv[i], STDOUT\_FILENO, &progress[i]);

fflush(stdout);

exit(EXIT\_SUCCESS);

}

}

int all\_done;

do {

all\_done = 1;

for (int i = 1; i < argc; ++i) {

if (progress[i] < 100) {

all\_done = 0;

}

display\_progress\_bar(argv[i], progress[i]);

fflush(stdout);

}

usleep(100000);

} while (!all\_done);

for (int i = 1; i < argc; ++i) {

wait(NULL);

}

munmap(progress, shm\_size);

printf("\n所有文件处理完毕\n");

return EXIT\_SUCCESS;

}

## 3.2 解压

利用archive库实现。

int unarchive(const char \*archivefile, int \*progress) {

struct archive \*a;

struct archive \*ext;

struct archive\_entry \*entry;

int flags;

int r;

flags = ARCHIVE\_EXTRACT\_TIME;

a = archive\_read\_new();

archive\_read\_support\_format\_archive(a);

ext = archive\_write\_disk\_new();

archive\_write\_disk\_set\_options(ext, flags);

archive\_write\_disk\_set\_standard\_lookup(ext);

if ((r = archive\_read\_open\_filename(a, archivefile, 10240))) {

fprintf(stderr, "无法打开文件 %s: %s\n", archivefile, archive\_error\_string(a));

return -1;

}

char \*dirname = create\_unique\_dir(archivefile);

if (!dirname) {

archive\_read\_close(a);

archive\_read\_free(a);

return -1;

}

int total\_files = 0;

while (archive\_read\_next\_header(a, &entry) == ARCHIVE\_OK) {

total\_files++;

archive\_read\_data\_skip(a);

}

archive\_read\_close(a);

archive\_read\_free(a);

a = archive\_read\_new();

archive\_read\_support\_format\_archive(a);

if ((r = archive\_read\_open\_filename(a, archivefile, 10240))) {

fprintf(stderr, "无法打开文件 %s: %s\n", archivefile, archive\_error\_string(a));

return -1;

}

int processed\_files = 0;

while (archive\_read\_next\_header(a, &entry) == ARCHIVE\_OK) {

const char \*currentFile = archive\_entry\_pathname(entry);

char newFilePath[MAX\_FILENAME];

snprintf(newFilePath, MAX\_FILENAME, "%s/%s", dirname, currentFile);

archive\_entry\_set\_pathname(entry, newFilePath);

//printf("正在解压: %s\n", currentFile);

archive\_write\_header(ext, entry);

archive\_copy\_data(a, ext);

archive\_write\_finish\_entry(ext);

processed\_files++;

\*progress = (processed\_files \* 100) / total\_files;

}

archive\_read\_close(a);

archive\_read\_free(a);

archive\_write\_close(ext);

archive\_write\_free(ext);

return 0;

}

## 3.3 解压

基于美观与个性化设计考虑，更加直观地显示操作进度。

void display\_progress\_bar(const char \*filename, int progress) {

printf("\r%s: [", filename);

int pos = (progress \* PROGRESS\_BAR\_WIDTH) / 100;

for (int i = 0; i < PROGRESS\_BAR\_WIDTH; ++i) {

if (i < pos) {

printf("=");

} else if (i == pos) {

printf(">");

} else {

printf(" ");

}

}

printf("] %d%%", progress);

fflush(stdout);

}

# 4 总结

**优点1多格式支持**：代码支持多种常见的压缩格式（如zip、tar、tar.gz、tar.xz、xz、gz、7z），能够满足大多数解压需求。

**优点2进度显示**：提供了实时的进度显示功能，用户可以直观地看到解压进度。

**优点3并行处理**：通过创建子进程实现并行处理多个文件，提升了解压效率。

**优点4动态目录创建**：为每个压缩文件创建唯一的解压目录，避免文件冲突。

**缺点：**缺少一定的大当量文件测试。