华南师範大學
SOUTH CHINA NORMAL UNIVERSITY

本科毕业论文

论文题目：短的论文题目一行
长的论文题目两行

指导老师：指导老师

学生姓名：你的名字

学　　号：**20152100001**

学　　院：计算机学院

专　　业：网络工程

班　　级：网络工程 7 班

摘　要

分析传输层协议中使用的零轮往返时长协议存在的重放攻击，编程实现简单传输层协议，重现相关重放攻击，并分别实现四种方法抵抗重放攻击。方法一：检查客户端发送零轮往返的 Ticker 是否过期。方法二：检查客户端发送零轮往返的 Ticker 是否被使用。方法三：服务器只接受零轮往返幂等性的请求。方法四：添加 Early-Data 请求头，告知服务器此请求是早期数据请求，服务器根据实际情况判断是否接受早期数据。进过测试分析得出，方法一、方法二和方法四组合能达到最优效果，保证服务器能正常处理早期数据的同时不受零轮往返时长重放攻击的影响。

关键词: 传输层协议；零轮往返时长；早期数据；重放攻击

Abstract

A replay attack is a form of network attack in which a valid data transmission is maliciously or fraudulently repeated or delayed. It is mainly used in the identity authentication process and destroys the correctness of authentication. In this paper, I analyze the replay attacks that exist in the zero round-trip time protocol used in the transport layer protocol. Programming to implement a simple transport layer protocol, then I reproduce the relevant replay attacks by programming and implement four methods to resist the replay attack protocol. The methods are as follows. First, examine whether the Ticker sent by the client to the zero round trip expires. Second, examine whether the Ticker sent by the client to the zero round trip is used. Third, make the server only accept zero rounds of idempotent requests. Fourth, add an Early-Data request header to inform the server that this request is an early data request, allowing the server to determine whether to accept early data based on actual conditions. After testing and analysis, I conclude that the combination of the first, the second and the fourth method can achieve the best effect, ensuring that the server can process the early data normally without being affected by the zero round-trip time replay attack.

Keywords: 0-RTT, TLS 1.3, Early Data, Replay Attacks

目　　录

摘要 I

Abstract II

目录 III

第一章　引言 1

1.1　选题背景与意义 1

1.2　国内外研究现状和相关工作 2

1.3　研究内容及主要贡献 3

1.4　论文章节安排 3

第二章　实现简单传输层协议 5

2.1　开发环境概述 5

2.2　开发语言概述 5

2.3　功能介绍 6

2.4　密钥导出算法 7

2.5　对称加密算法 8

第三章　分析零轮往返时长协议中的重放攻击 11

3.1　传输层协议 1.3 版本简介 11

3.2　零轮往返时长协议发生条件 12

3.3　零轮往返时长协议介绍 14

3.4　重放攻击原理 15

3.5　防御一般性重放攻击 16

3.6　防御零轮往返时长协议重放攻击 17

第四章　测试与结果 21

第五章　总结与展望 23

5.1　总结 23

5.2　展望 24

附录 A　HKDF 25

附录 B　AEAD 26

附录 C TLS 1.3 状态机 27

C.1 客户端状态机 27

C.2 服务器状态机 28

附录 D 密钥导出流程图 29

参考文献 30

致谢 32

第一章　引言

1.1　选题背景与意义

安全传输层协议 (Transport Layer Security，下面简称 TLS) 每天被全球数百万用户使用，作为互联网安全的核心构建块，不仅应用到浏览器的 HTTPS 协议中，还有其他应用层协议也使用了 TLS，比如 SSH、WSS 协议等。由于 TLS 1.2 及以下版本中的各种安全缺陷和设计缺陷 [1], [2], 会受到降级攻击、中间人攻击、BEAST 攻击、POODLE 攻击等，最为严重的 openssl 心脏出血漏洞 [3]，属于实现上的漏洞而不是协议上的漏洞，还有 TLS 1.2 的完整握手需要两轮往返时间，耗时长，即使使用会话恢复也需要一轮往返时间。

无论是基于实现还是基于规范，都促使 TLS 工作组在起草下一版协议时采用“部署前分析”设计范例 [4]。TLS 协议主要目标是在两个通信应用程序之间提供数据保密性和数据完整性。IETF(国际互联网工程任务组) 用了超过四年时间，起草了 28 份草案后，于 2018 年 8 月发布 TLS 1.3 [5] 最终版本 (RCF 8446)。相比较于 TLS 1.2，TLS 1.3 删除未使用或者不安全的功能，并且加密了更多的握手信息，减少了握手延迟。TLS 协议的主要目标是在通信双方提供安全信道，包括对服务器和客户端的身份验证，信道上传输数据的机密性，保证通信双方传输内容的完整性。非对称加密算法中：在 TLS 1.3 中密钥交换默认使用椭圆曲线加密法，删除静态 RSA 密钥交换。对称加密算法上：删除 CBC 模式密码、RC4 流密码，保证信息机密性和完整性使用 AEAD 算法 [6]，是 TLS 1.3 中唯一保留的对称加密方式，只有 AES-GCM 和 ChaCha20-Poly1305 两种，更安全和难以破解。删除 SHA-1 哈希函数，哈希算法使用具有更长密钥的 SHA-2 哈希函数，其中包括 SHA-256 和 SHA-384。

此外，TLS 1.3 使用新的 PSK 密钥协商。TLS 1.3 不仅在安全性上有提升，速度上也提升不少。TLS 1.3 相比较于 TLS 1.2 的两次往返才能建立连接的完整握手，针对 TLS 1.2 中两次往返耗时长，速度较慢比较慢，为了有更快是网络访问速度，TlS 1.3 完整握手使用了也减少到一次往返和会话恢复的零轮往返时长协议握手，大大提升连接速度，比 TLS 1.2 减少了一个握手往返，仅在传输层上速度将提升一倍。

虽然 TLS 1.3 和零轮往返时长协议不能减少传输的往返延迟，但可以减少建立 HTTPS 连接所需的往返次数，从而减少握手花费的时间。有更快的握手速度固

然是好，但同时不能忽略一些安全问题。利用 WebSocket 编程简单实现 TLS 协议，分析和重现 TLS 1.3 协议中 0—RTT 的重放攻击，并在后台服务器中实现抵抗重放攻击方法，测试抵抗重放攻击的实用性。

1.2 国内外研究现状和相关工作

有关 TLS 1.3 的讨论，多数停留在非正式发布前 [7] [8] [9] 的草案，基于 DH 密钥交换的零轮往返时长协议，但在正式发布前已经删除，最终使用的是基于 PSK 的握手恢复。在国内，只能在期刊上阅读到少量关于 TLS 1.3 的文章，而且文章内容讲述过于简单，不够全面，关于 TLS 1.3 协议最重要的部分零轮往返时长协议没有更多讲解，国内知名通信软件微信，参考 TLS 1.3 实现的安全通信协议 MMTLS [10]，其中实现零轮往返时长协议，比较于 TLS 协议，MMTLS 协议由于微信客户端每个人可用，于是删除了客户端认证相关的功能，同时在微信客户端程序中内置服务器的签名公钥，握手中不在需要进行服务器认证，减少发送的流量，关于 MMTLS 抗重放攻击，根据微信特有的后台架构，提出了基于客户端和服务器端时间序列的防重放策略，保证超过时间的重放包能被服务器拒绝，通过由 Proxy 层和 Logic 框架协同控制。

支持 TLS 1.3 的代码库，最广泛的当然有 Openssl，Google 的 boringssl、guntls 等。国外相关工作，谷歌在基于 UDP 协议上实现的 QUIC Crypto 协议 [11] 中首次实现了零轮往返时长协议，由于 QUIC 更早的使用零轮往返时长协议，实现的标准也提供给 TLS 1.3 作为参考，但到了 TLS 1.3 正式版本发布，QUIC 反而会基于 TLS 1.3，并在以后的 HTTP/3.0 中使用，促进网络协议的发展。Facebook 使用 C++14 标准实现强大，高性能的 TLS 库，代码库命名为 Fizz，在 QUIC 基础改做出改动，在手机 APP 上实现零轮往返，实现更快的连接速度，并且有效地处理安全性问题，实现部署零轮往返时长协议，发现建立连接所需的时间降低了 41%。

在密钥交换的同时减少延迟开销已成为学术界和工业界的密钥交换)(key exchange, KE) 协议的主要设计目标。在这方面特别感兴趣的是零轮往返时长协议，其允许客户端在零往返时间中发送加密数据，从而最小化等待时间。比如 Google 的 QUIC 协议和 TLS 1.3。零轮往返密钥交换的主要挑战是为协议中发送的第一个加密数据，称为早期数据，实现前向保密和防止重放攻击的安全性。有不少说法称不可能为此消息实现前向保密，因为用于加密早期数据的密钥必须依赖于接收者的非短暂密钥。如果接收者的非密钥在以后泄露给攻击者，攻击者应该可以执行与实际会话中的接收者相同的计算来计算会话密钥。

在相关研究中，表明不能为早期数据提供前向保密性是错误的想法。研究人员构建了第一个零轮往返时长协议，利用一种可穿透的密钥封装方案，它为所有传输的有效负载消息提供完全的前向保密，并自动适应重放攻击。由于在没有先前连接的信息的情况下，是不可能在 0-RTT 中进行身份验证和建立加密密钥，因此 0-RTT 密钥交换协议必须利用在某些先前通信中获得的密钥材料来建立 0-RTT 密钥。在早期版本的 QUIC 中使用的一种非常常见的方法是基于 DH 密钥交换，但在正式发布的 TLS 1.3 中采用的是从预共享对称密钥 (PSK) 中导出 0-RTT 密钥。

1.3 研究内容及主要贡献

分析 TLS 1.3 协议，利用 WebSocket 简单实现握手协议和记录层协议，重现 TLS 1.3 中 0-RTT 的重放攻击。实现对抗 0-RTT 重放的方法并测试效果。得到以下结果:

- TLS 1.3 比旧版本更安全可靠
- 0-RTT 握手在连接时速度更快
- 相关实现可以正确抵抗 0-RTT 重放攻击

1.4 论文章节安排

第一章引言
第二章实现简单 TLS 1.3 协议
第三章分析零轮往返时长协议中的重放攻击
第四章测试和实验结果
第五章总结与展望

第二章　实现简单传输层协议

完整的 TLS 1.3 协议栈支持需要大量研究的开发工作，比如 openssl 库，单人开发几乎是不可能的，在此使用浏览器/服务器模式实现简单的 TLS 1.3 协议，包括握手协议和记录层协议 [12]。利用 WebSocket 实现密钥协商功能、参数协商功能、建立共享密钥功能。WebSocket 是 HTML5 提供的新功能，一种浏览器能与服务器之间进行全双工通信的网络技术，可以传输文本和二进制数据。

2.1　开发环境概述

1. 服务器端:

   操作系统: OS X El Capitan

   服务器: Node.js v10.15.3

   数据库: Mongodb 和 Redis

   开发工具: Visual Studio Code

   浏览器: Chrome v73

2. 客户端:

   开发工具: Wepack, Visual Studio Code

   浏览器: Chrome v73

2.2　开发语言概述

本次开发讲使用 HTML5、CSS3、JavaScript 等基本编程语言。HTML5 为我提供不少的浏览器接口，localStorage，Cookie 等存储方式，WebSocket 协议能使服务器主动推送消息到客户端，完成 TLS 1.3 握手功能。HTML5 为网页提供最基本骨架，页面的元素，让浏览器应用开发更加多样化，功能更强大的技术。各大浏览器支持情况良好。CSS3 作为 HTML5 的化妆师，提供页面美化和页面布局，样式化和排版前端网页，例如控制页面字体大小、颜色、间距等。CSS3 选择器需要结合 HTML5 使用，在 HTML5 元素中使用对应的 CSS 选择器。JavaScript 是一种面向对象的动态语言，提供浏览器和用户进行页面交互的操作，是开发中的核心技术，在编程语言中排行前 10。比较轻量级，插入到 HTML5 页面中，可以通过大多数浏览器解析执行。

服务器开发使用 Node.js，实际上它是对 Google Chrome V8 引擎进行了封装，它主要用于创建快速的、可扩展的网络应用。Node.js 采用事件驱动和非阻塞 I/O 模型，使其变得轻量和高效。对比解析运行在 Chrome 浏览器上的 JavaScript 代码，Chrome 浏览器就是 JavaScript 代码的解析器，当 JavaScript 代码运行在服务器上时，Node.js 就是 JavaScript 代码的解析器，存在于服务器端的 JavaScript 代码由 Node.js 来解析和运行。JavaScript 解析器用于提供 JavaScript 代码运行的一种环境，浏览器是 JavaScript 运行的一种环境，不同的浏览器有不同的解析引擎，浏览器为 JavaScript 提供了操作文档对象模型和浏览器窗口对象等的接口。Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境，提供了操作系统文件、创建 HTTP 服务、创建 TCP/UDP 服务等的接口，Node.js 的包管理器 npm，是全球最大的开源库生态系统。

2.3 功能介绍

实现握手协议，记录层协议。使用 WebSocket 交换客户端和服务器端的密钥参数，主要实现 AEAD 算法: AES-256-GCM, ChaCha20-Poly1305。HKDF [13] 密钥导出算法。编程实现 FRC8446 中客户端和服务器端的状态机。完整的一轮往返握手和零轮往返，一轮往返握手为之后建立零轮往返建立基础，用于实现重放攻击。

密钥交换信息。客户端发送的 ClientHello 中提供：

1. legacy_version: 在 TLS 1.3 版本中必须设置成 0x0303。
2. random: 安全随机数生成器产生的 32 字节随机数。
3. legacy_session_id: 兼容模式下，这个值必须是非空，一个不可预测的值。
4. cipher_suites: 客户端支持的加密套件，TLS 1.3 中只支持 5 种加密套件。
5. legacy_compression_methods: 必须设置为 0 的一个字节。
6. extensions: 拓展

服务器发送的 ServerHello 中提供：

1. legacy_version: 在 TLS 1.3 版本中必须设置成 0x0303。
2. random: 安全随机数生成器产生的 32 字节随机数。
3. legacy_session_id_echo: 兼容模式下，这个值必须是非空，一个不可预测的值。

4. cipher_suites: 从客户端支持的加密套件中选择加密套件。

5. legacy_compression_methods: 必须设置为 0 的一个字节。

6. extensions: 拓展。TLS 1.3 版本中必须包含 supported_versions 扩展

服务器参数和认证消息这里不展开详细说明。Certificate: 将证书链发送给对方，当约定的密钥交换方法是用证书进行认证的时候，服务器就必须发送 Certificate 消息，当且仅当客户端通过发送 CertificateRequest 消息请求认证客户端时，客户端必须发送 Certificate 消息，当客户端没有合适的证书时，必须发送不含证书的 Certificate 消息。Certificate Verify: 此消息用于证明发送方拥有其证书对应的私钥，必须在 Certificate 消息之后立即发送，并且紧接着在 Finished 消息之前。Finished: 提供握手和密钥的身份验证，Finished 消息的接受者必须验证内容是否正确，如果不正确，必须使用 decrypt_error alert 消息终止连接。End of Early Data: 如果服务器在 EncryptedExtensions 中发送了 early_data 扩展，则客户端必须在收到服务器的 Finished 消息后发送 EndOfEarlyData 消息。

2.4 密钥导出算法

在 TLS 1.3 中，不再使用 PRF 算法，而是采用更标准的 HKDF 算法来进行密钥的推导。而且在 TLS 1.3 中对密钥进行了更细粒度的优化，每个阶段或者方向的加密都不是使用同一个密钥。TLS 1.3 在 ServerHello 消息之后的数据都是加密的，握手期间服务器给客户端发送的消息用 server_handshake_traffic_secret 通过 HKDF 算法导出的密钥加密的，Client 发送给 Server 的握手消息是用 client_handshake_traffic_secret 通过 HKDF 算法导出的密钥加密的。这两个密钥是通过 Handshake Secret 密钥来导出的，而 Handshake Secret 密钥又是由 PreMasterSecret 和 Early Secret 密钥导出，然后通过 Handshake Secret 密钥导出主密钥 Master Secret。

再由主密钥 Master Secret 导出这几个密钥：client_application_traffic_secret：用来导出客户端发送给服务器应用数据的对称加密密钥。server_application_traffic_secret：用来导出服务器发送给客户端应用数据的对称加密密钥。resumption_master_secret：用来生成 PSK。最终 server_handshake_traffic_secret、client_handshake_traffic_secret、client_application_traffic_secret、server_application_traffic_secret 这 4 个密钥会分别生成 4 套 write_key 和 write_IV 用于对称加密。如果用到 early_data，还需要 client_early_traffic_secret，它也会生成 1 套 write_key 和 write_IV 用于加密和解密 0-RTT 数据。Key Derivation Function (KDF) 是密码学系统中必要的组件。它的

目的是把一个 key 拓展成多个从密码学角度来上说是安全的 key。

TLS 1.3 使用的是 HMAC-based Extract-and-Expand Key Derivation Function 即 HKDF 函数，HKDF 根据 extract-then-expand 设计模式，即 KDF 有 2 大模块。第一个阶段是将输入的 key material 进行”extracts”，得到固定长度的 key，然后第二阶段将这个 key ”expands” 成多个附加的伪随机的 key，输出的 key 的长度和个数，取决于指定的加密算法。由于 extract 流程不是必须的，所以 expand 流程可以独立的使用。HMAC 的两个参数，第一个是 key，第二个是 data。data 可以由好几个元素组成，一般用 | 来表示

经过密钥协商得出来的密钥材料的随机性可能不够，协商的过程能被攻击者获知，需要使用一种密钥导出函数来从初始密钥材料（PSK 或者 DH 密钥协商计算出来的 key）中获得安全性更强的密钥。HKDF 正是 TLS 1.3 中所使用的这样一个算法，使用协商出来的密钥材料和握手阶段报文的哈希值作为输入，可以输出安全性更强的新密钥。在 TLS 1.2 中使用的密钥导出函数 PRF 实际上只实现了 HKDF 的 expand 部分，并没有经过 extract，而直接假设密钥材料的随机性已经符合要求。因为 TLS 1.3 对密钥材料进行 extract_then_expand，所以这也是为什么 TLS 1.3 比 TLS 1.2 在安全性上更上一层楼的原因。TLS 1.3 中的所有密钥都是由 HKDF-Extract(salt, IKM) 和 Derive-Secret(Secret, Label, Messages) 联合导出的。其中 Salt 是当前的 secret 状态，输入密钥材料 (IKM) 是要添加的新 secret 。在 TLS 1.3 中，两个输入的 IKM 是: PSK 或者 (EC)DHE 共享的 secret。一旦计算出了从给定 secret 派生出的所有值，就应该删除该 secret。

2.5 对称加密算法

TLS 1.3 中使用的 AEAD 算法有 AES-256-GCM，AES-128-GCM 和 ChaCha20-Poly1305，AEAD 算法操作有 4 个输入：1. 要加密的明文、2. 密钥、3. 一个独特的初始化值 - IV。在使用相同密钥调用加密操作之间必须是唯一的，否则密码的保密性将完全受到损害、4. 可选部分，一些其他非秘密的附加数据，此数据不会被加密，但会进行身份验证。在数据被加密之后，加密算法使用密钥（以及可选地加入 IV）来生成辅助密钥。辅助密钥用于生成 AD 的密钥散列，密文和每个密钥的各个长度。ChaCha20-Poly1305 中使用的散列函数是 Poly1305，而在 AES-GCM 中，散列函数用的是 GHASH。最后一步是获取哈希值并对其进行加密，生成最终的 MAC（消息认证码）并将其附加到密文。解密操作与加密相反。它采用相同的密钥和 IV 并生成密文和 AD 的 MAC，类似于加密的方式。然后它读取密文之后

附加的 MAC，并比较两者。MAC 值有任何差异都意味着密文或 AD 被篡改，并且它们应该被认为不安全而丢弃。如果两者匹配，则执行解密操作，恢复原始明文。AEAD 将两种算法 - 加密算法和 MAC 算法组合成一个算法，具有可证明的安全性。当 AES_GCM 被破解时，ChaCha20-Poly305 是一种候选算法。数字 20 表示它总共重复 20 轮加密操作。它从递增的计数器生成伪随机比特流，然后用明文对该流进行“异或”以对其进行加密（或者用密文进行“异或”以解密）。

不需要提前知道明文来生成流，所以这种方法既可以非常高效又可以并行化。ChaCha20 是一个 256 位密钥加密算法，Poly1305 可以与任何加密或未加密的消息一起使用，以生成密钥认证令牌。这种令牌的目的是保证给定消息的完整性。对于 AES 块加密算法，在某些硬件上使用 AES-NI 加速指令，可以运行非常快，如果没用加速指令，单纯使用软件运行，性能会很低。而流密码 ChaCha20，则相反，软件实现性能很高，由于大部分移动设备没有 AES-NI 加速指令，运行 AES 会比较慢。ChaCha20-Poly1305 流密码算法来了，除了安全性外，它在移动设备上运行的性能较高。

CHACHA20 流加密算法。其原理和实现大致可以分成如下两个步骤：

1. 基于输入的对称秘钥生成足够长度的 keystream

2. 将上述 keystream 和明文进行按位异或，得到密文

ChaCha 密钥为 256 位（K =（k0，k1，k2，k3，k4，k5，k6，k7，以 32 位密钥运行。这个输出块为 512 位，用于密钥流（Z），以及与明文流进行异或运算。初始状态包含 16 个 32 位值，组成 4x4 矩阵，具有 128 位的常量（0x61707865,0x3320646e，0x79622d32,0x6b206574）256 位的密钥（k0，k1，k2，k3，k4，k5，k6，k7），32 位的计数器（c）和 96 位的 nonce（N0，N1，N3）。ChaCha20 一共进行 20 轮加密操作，10 轮列操作，10 轮对角线操作。初始化矩阵加 20 轮操作之后的矩阵等到密钥流矩阵，和明文进行异或运算得出密文。由于计数器是 32 位，理论上可以生成 2 5̃12 bit（256GB）的密钥流，所以一般长度的信息加密完全足够。解密操作和加密操作一样，接收方与发送方生成一样密钥流矩阵，与密文异或解密得到明文。

ChaCha20-Poly1305 优势 [14]，[15] Google 推出新的加密套件并在所有移动端的 Chrome 浏览器上优先使用原因：ChaCha20-Poly1305 避开了现有发现的所有安全漏洞和攻击；ChaCha20-Poly1305 针对移动端设备大量使用的 ARM 芯片做了优化，能够充分利用 ARM 向量指令，在移动设备上加解密速度更快、更省电；Poly1305 输出只有 16 字节，更加节省带宽

第三章　分析零轮往返时长协议中的重放攻击

3.1　传输层协议 1.3 版本简介

TLS 协议由两层协议组成：TLS 记录协议和 TLS 握手协议。TLS 1.3 协议只支持 (EC)DHE 的密钥协商算法，删除的 RSA 密钥协商算法和静态 DH 密钥协商算法，更好地提供向前保密性，防止心脏出血攻击，移除所有有安全隐患的密码和密码模式，并且唯一允许的对称加密是 AEAD。除了 ClientHello 和 ServerHello 报文外，其他握手报文全部加密，这在 1.2 版本并没有。服务器现在签署了整个握手，计算 HMAC 的时候使用全部握手信息，防止降级攻击。

并且不再允许对加密报文进行压缩、禁止重协商。TLS 1.3 握手协议用于协商连接的安全参数，包括密钥协商，参数协商，建立共享密钥，负责协商使用的 TLS 版本、加密算法、哈希算法、密钥材料和其他与通信过程有关的信息，对服务器进行身份认证，对客户端进行可选的身份认证，最后对整个握手阶段信息进行完整性校验以防范中间人攻击，是整个 TLS 协议的核心。

握手有三个阶段: 第一阶段：密钥交换。第二阶段：服务器端参数。第三阶段：认证。TLS 1.3 协议握手有两种模式：一种是 1-RTT 握手，另一种是 0-RTT 握手。在 1-RTT 模式下客户端和服务器端只需要进过一次往返就能完成协商连接需要用到的安全参数，并且完成身份验证。在 0-RTT 模式中，客户端通过发送 PSK 和 early_data 完成握手和身份验证，其中 PSK 用于恢复主密钥，early_data 的是经过加密的应用层数据，比如发送的一次 GET 请求。客户端和服务器端在没有数据往返情况下从 PSK 中导出 early_data 解密密钥，解密 early_data 数据。0-RTT 模式的握手需要 3 个条件：

1. 客户端和服务器端不是第一次握手连接，并且之前的连接服务器端发送了 Session Ticket，Session Ticket 中包含愿意接受 early_data 的 max_early_data_size。

2. 不是第一次握手时，客户端发送了 PSK 和 early_data。

3. 服务器端读取客户端发送过来的 early_data。

TLS 1.3 记录协议负责接收要传输的应用层数据，将数据分段为合适的长度，输入密钥、随机字符串，使用 AEAD 算法加密分段后的数据，添加记录层头部并传输最后处理结果。收到的数据经过验证，解密，重新组装，然后交付给更上层的协议。信息块分段为 TLSPlaintext 记录，TLSPlaintext 中包含 214 字节或更少字节块的数据。记录保护功能将 TLSPlaintext 结构转换为 TLSCiphertext 结构。去除

保护功能和保护功能互为逆过程。其中使用 Pre-Record Nonce 分别维护 64 位序列号以读取和写入记录，Pre-Record Nonce 是按网络字节顺序编码的 64 位序列号和 client_write_iv 或 server_write_iv 异或得到。在读取或写入每个记录之后，适当的序列号加 1。每个序列号在连接开始时和每次更改密钥时都设置为零，在特定流量密钥下传输的第一条记录必须使用序列号 0。因为序列号的大小是 64 位，所以它们不应该换行。

如果 TLS 实现方需要 wrap 序列号，它必须重新生成密钥或终止连接。每个 AEAD 算法将为 Per-record 的随机数指定一系列可能的长度，从 N_MIN 字节到输入的 N_MAX 字节 [RFC5116]。对于 AEAD 算法，TLS 的 Per-record 随机数 (iv_length) 的长度设置为 8 字节和 N_MIN 中的较大者 (参见 [RFC5116]，第 4 节)。其中 N_MAX 小于 8 个字节的 AEAD 算法不得与 TLS 一起使用。所有加密的 TLS 记录都可以被填充，从而扩大 TLSCiphertext 的大小。这种做法允许发送者对攻击者隐藏流量大小。生成 TLSCiphertext 记录时，实现方可以选择填充。未填充的记录只是填充长度为零的记录。填充是在加密之前附加到 ContentType 字段的一串零值字节。实现方必须在加密之前将填充的八位字节全部设置为零。

TLS 警报协议发送的 Alert 消息传达警报的描述以及在先前版本的 TLS 中传达消息严重性级别的遗留字段。警报分为两类：关闭警报和错误警报。TLS 提供 Alert 内容类型用来表示关闭信息和错误。与其他消息一样，Alert 消息也会根据当前连接状态的进行加密。收到错误警报后，TLS 实现方应该向应用程序表示出现了错误，并且不允许在连接上发送或接收任何其他数据。服务器端和客户端必须删除的旧连接中建立的秘密值和密钥。客户端和服务器端必须共享连接结束的状态，以避免截断攻击。任何一方都可以通过发送 close_notify 警报来发起其连接写入端的关闭。收到关闭警报后收到的任何数据都必须被忽略。当检测到错误时，检测的这一方，向其对等方发送消息。在传输或收到致命警报消息时，双方必须立即关闭连接，而不发送或接收任何其他数据。

3.2 零轮往返时长协议发生条件

在 TLS 1.3 版本中，零轮往返时长需要发生在一轮往返握手之后，一轮往返握手又分为两种：完整的握手和不发送 early_data 的 PSK 握手。完整握手中客户端发送 ClientHello，Extension 带上客户端所有支持的椭圆曲线类型，并且计算每个椭圆曲线的公钥和私钥，私钥缓存，公钥放在 Extension 中的 key_share 中发送给服务器。服务器端接收到客户端发送的 ClientHello 后，选择客户端支持的加密套

件，如椭圆曲线参数 (曲线、基点)，计算自己的公钥和私钥，然后从客户端发送过来 ClientHello 中的 key_share 拓展中选择对应的椭圆曲线公钥，并用私钥计算预主密钥。

服务器端回复 ServerHello、Certificate、Certificate Verify、Finished 报文，其中 ServerHello 中的 key_share 拓展发送选择的椭圆曲线和公钥。客户端收到 ServerHello 后，从 key_share 拓展中取出服务器发送的公钥，与自己的私钥计算预主密钥，得到预主密钥，通过一系列的 HKDF 函数计算导出其他主密钥。因为客户端和服务器端 early_secret 和协商出来的预主密钥相同，因此所有后续进过 HKDF 函数导出的对应的密钥都是相同的。发送 Finished 消息后，完成了整个握手过程。通过 master_secret 和整个握手的摘要，计算 resumption_secret。服务端收到客户端的 Finished 消息后，通过验证后，同样计算 resumption_secret。握手完成之后，服务器可以在以后的任意时刻发生 NewSessionTicket，NewSessionTicket 使用 server_application_traffic_secret 加密。

在加密的 Ticket 中，相比 TLS 1.2，包含了当前的创建时间，因此可以方便的配置和验证 Ticket 的过期时间。在 PSK 握手中，一旦初次握手已经完成，服务器端就能给客户端发送一个与独特密钥对应的 PSK，这个密钥来自初次握手。客户端可以在将来的握手中使用该 PSK。如果服务器端接受了客户端发送的 PSK，则新连接的安全上下文与原始连接相关联，并且使用从初始握手导出的密钥来引导加密状态而不是完全握手。PSK 可以与（EC）DHE 密钥交换一起使用，以提供与共享密钥相结合的前向保密，或者可以单独使用，以牺牲保密性为代价。

当服务器端通过 PSK 进行身份验证时，它不会发送 Certificate 或 CertificateVerify 消息。当客户端通过 PSK 恢复时，它还应该为服务器端提供一个 key_share 扩展，以允许服务器端拒绝恢复使用之前的连接状态，如果需要，可以恢复到完全握手。服务器端用 pre_shared_key 扩展进行响应以协商使用 PSK 密钥建立，并且可以用 key_share 扩展来响应（EC）DHE 密钥建立，从而提供前向保密。客户端为了使用 PSK 进行握手，必须发送一个 psk_key_exchange_modes 扩展。这个扩展语意是客户端仅支持使用具有这些模式的 PSK。这就限制了在这个 ClientHello 中提供的 PSK 的使用，也限制了服务器端通过 NewSessionTicket 提供的 PSK 的使用。0-RTT 降级到 1-RTT：服务器端拒绝 PSK 握手，ServerHello 中不加入 pre_shared_key，服务器端拒绝 early_data。ServerHello 中加入了 pre_shared_key 但是 Encrypted Extension 报文中不加入 early_data

3.3 零轮往返时长协议介绍

零轮往返时长协议 (下面简称 0-RTT) 是 TLS 1.3 协议的一个重要的新功能，应用层数据加密后和客户端的 ClientHello 一起发送到服务器，不需要客户端和服务器的往返协商密钥，能提升页面加载速度。其中 0-RTT 握手的的前置条件：

1. 服务器在前一次完整握手中，发送了 NewSessionTicket，并且 NewSessionTicket 中存在 max_early_data_size 扩展表示愿意接受 early_data。如果没有这个扩展，0-RTT 无法开启。

2. 在 PSK 会话恢复的过程中，ClientHello 的扩展中配置了 early_data 扩展，表示客户端想要开启 0-RTT 模式。

3. 服务器在回复 0-RTT 的 ClientHello 中发送的 Encrypted Extensions 消息中携带了 early_data 扩展表示同意读取 early_data。

0-RTT 的握手发送条件：

1. 不是第一次握手的连接。

2. 非第一次握手时：客户端发送了 PSK 和 early_data 拓展。

3. 服务器读取 early_data。0-RTT 模式开启成功。当客户端和服务器都支持 TLS 1.3 时，客户端会将收到服务器发送过来的 NewSessionTicket，和通过自己发送 Finished 后计算得到的 Resumption Secret，两者一起组成 PSK。

客户端收到 NewSessionTicket 消息后，将收到的 Ticket 和客户端本地发送 Finished 消息后计算的 resumption_secret，两者一起组成了 PSK，将该 PSK 缓存在本地，ServerName 作为缓存的 Key，缓存时间不可以超过 7 天。当进行会话恢复时，客户端在本地缓存中查找 ServerName 对应的 PSK，用在发送的 ClientHello 的 PSK 扩展中，PSK 拓展会包含两部分：Identity 和 Binder。其中 Identity 是服务器发送过来 NewSessionTicket 中加密的 Ticket，Binder 是一个 HMAC，是从之前客户端发送 Finised 计算的 Resumption Secret 导出 early_secret，进而导出 binder_key 和 binder_mac_key，使用 binder_mac_key 对不包含 PSK 部分的 ClientHello 计算 HMAC。

发送 ClientHello 后，使用 resumption_secret 导出的 early_secret 加密 early_data 后发送。服务器会从 ClientHello 中验证 Binder，当验证通过时，使用 PSK 通过 HKDF 函数导出密钥，然后解密刚才发过来的 early_data。服务器收到客户端的 ClientHello 之后，生成 key_share，检查 ClienHello 的 PSK 扩展，解密 Ticket，此 Ticker 在客户端看不有什么意义，但服务器能正确解读，继续查看该 Ticket 是否过期，检查客

户端发送的版本算法等协商结果是否可用，然后使用 Ticket 中的 resumption_secret 计算 ClientHello 的 HMAC，检查 Binder 是否正确。验证完 Ticket 和 Binder 之后，和客户端一样，从 resumtion_secret 中导出 EarlyData 使用的密钥，然后解密客户端发送过来的 EarlyData。

发送 ServerHello 中表示使用了 PSK 进行握手，以及哪个 PSK。在服务端，TLS 1.3 只使用过去的 resumption_secret 导出 early_data 的密钥，但是之后的握手和通信的主密钥会和临时 DH 密钥一起导出。如果服务器在 Encrypted Extensions 中发送了 early_data 扩展，则客户端必须在收到服务器的 Finished 消息后发送 EndOfEarlyData 消息。如果服务器没有在 EncryptedExtensions 中发送 early_data 扩展，那么客户端绝不能发送 EndOfEarlyData 消息。收到客户端发送的 EndOfEarlyData 后，切换到应用程序密钥，此消息表示已传输完了所有 0-RTT application_data 消息 (如果有)。服务器不能发送此消息，Client 如果收到了这条消息，那么必须使用”unexpected_message” alert 消息终止连接。这条消息使用从 client_early_traffic_secret 中派生出来的密钥进行加密保护。至此完成了该 0-RTT 会话恢复的过程。

3.4 重放攻击原理

当攻击者窃听并且截获安全网络通信，然后欺骗性地延迟或重新发送被攻击者在通信中已经使用过的正确消息，消息被正确加密，服务器接收到之后正确解密，就会发生重放攻击。重放攻击的另一个危害是，攻击者在从网络捕获消息后甚至不需要使用高级技能来解密消息。只需重新发送整个消息内容，攻击就可以成功。除非服务器有缓解措施，否则受重放攻击的服务器会将攻击过程视为正确合法的消息。重放攻击的一个例子是消息由攻击者重放发送到网络，该消息先前由服务器授权用户发送。尽管消息可能已加密且攻击者可能无法获得实际密钥解密消息，但重放有效数据或登录消息可帮助攻击者获得对服务器的授权访问。重放攻击可以通过重放身份验证消息来访问资源，并且可能会混淆目标服务器。

攻击者虽然不知道 0-RTT 发送 Early Data 的意义，但可能知道它的意义，TLS 1.3 1-RTT 加密数据为了防止重放攻击，使用 AEAD 加密数据块的时候使用的 Nonce 都不同。在 0-RTT 的握手方式中，第一个加密的应用层数据和握手数据一起发送给服务器，对于第一个数据的防重放，服务器只能完全靠客户端发来的数据来判断是否重放，如果客户端发送的数据完全由自己生成，没有包含服务器参与的标识，那么这份数据是无法判断是否为重放数据包。TLS 1.3 给了一个思路来解决 0-RTT 跨连接重放的问题：在服务器保存一个跨连接的全局状态，每新建一

个连接都更新这个全局状态，那么 0-RTT 握手带来的第一个应用层数据也可以由这个跨连接的全局状态来判断是否重放。

针对 0-RTT 的重放攻击，攻击方式有两种：1. 通过简单拦截、复制 0-RTT 数据并发送来进行重放攻击。攻击者截获 0-RTT 消息并且保存下来，将整个 0-RTT 消息转发到服务器。服务器正确处理 PSK 解密出 Early Data，服务器执行响应的操作处理请求。2. 多个服务器负载均衡时，不维护一致的服务器状态，利用客户端重试行为使服务器接收多个 0-RTT 消息的副本。攻击者可以通过恶意 Javascript 代码在客户端的浏览器上重复发送 0-RTT 请求，在负载均衡时，将来自客户端的请求平均的转发到每个后台服务处理，没有共享 0-RTT 有效状态时，会导致一个或者多个服务重复处理 0-RTT 中数据请求，比如发生了多次转账操作。

3.5 防御一般性重放攻击

2012 年 Lan Sun 和 Zhao Luo 等人 [16] 提出了基于分组的算法防止重放攻击。防止重放攻击的最优技术之一是使用带有时间戳的服务器数字签名，可用于避免重放攻击的另一种技术是创建时间绑定的随机会话密钥。每个请求的一次性密钥也有助于防止重放攻击，并且经常用于银行上金钱的操作。用于对抗重放攻击的其他技术包括消息排序和不接受重复消息。

1. 使用一次性密码:

   一次性密码类似于会话令牌，因为密码在使用后或在很短的时间后到期。除会话外，它们还可用于验证单个事务。这些也可以在身份验证过程中使用，以帮助在彼此通信的双方之间建立信任。

2. 使用会话令牌标记加密消息:

   因为为客户端每次连接都会创建了唯一的随机会话令牌，因此先前的连接变得更难以复制。在这种情况下，攻击者将无法执行重播，因为在新的连接中，会话令牌会发生变化。服务器向客户端发送一次性令牌，客户端用它来转加密用户密码并将结果发送给服务器。例如，她将使用令牌来计算会话令牌的哈希函数，并将其附加到要使用的密码。同时，服务器使用会话令牌执行相同的计算。当且仅当客户端和服务器的值匹配时，登录成功。现在假设攻击者已经获取此会话令牌并尝试在另一个会话中使用它。服务器会发送一个不同的会话令牌，当攻击者回复服务器重放的值时，它将与服务器的计算不同，因此服务器知道不是客户端正常的请求。会话令牌应随机选择（通常使用伪

随机函数）。否则，攻击者可能会伪装成服务器，发送一些猜测的会话令牌，并欺骗客户端使用该令牌。然后，攻击者可以在未来某个时候重放消息（当先前猜测的会话令牌由服务器实际产生时），并且服务器将接受该重放消息。

3. 时间戳:

时间戳是防止重放攻击的另一种方法。应使用安全协议实现时间同步。例如，服务器定期广播他的时钟上的时间。当客户端想要向服务器发送消息时，在消息中包括对时间的最佳估计，该消息是被认证的。服务器只接受时间戳在合理范围误差内的消息。这种方案的优点是服务器不需要生成（伪）随机数，并且客户端不需要向服务器请求随机数。缺点是攻击者能足够快地执行重播攻击，服务器在合理时间限制内接收到，则会攻击可以成功。

3.6 防御零轮往返时长协议重放攻击

参考 Marc Fischlin 和 Felix Günther [17] 研究重放攻击下 0-RTT 的安全性，以及 RFC 中提到的抗重放攻击方法。编程实现相关方法措施抵抗重放攻击。第一个措施是检查 Ticker 是否过期。服务器在创建 Ticker 时候可以设置较短的有效期，保存创建此 Ticker 的时间，在 Redis 数据库中，以 Ticker 作为 Key，创建时间作为 Value 保存。接收到客户端发来的 Ticker 时，从 Redis 数据中获取创建时间，然后检查是否过期了，如果过期了就不处理 Early Data 中的数据，并且将握手降级到 1-RTT。服务器发送的 NewSessionTicket 中的 Ticket Lifetime 有效期最多是 7 天，但为了减少有效期，选择设置有效期为 5 分钟，使用时候单位转为秒，并且发送 Ticket Age Add 用于计算客户端发送的 Obfuscated Ticket Age 过期时间，计算为 Ticket Lifetime (转化为毫秒为单位) + Ticket Age Add 模 2^{23}。服务器接收到 0-RTT 时候，使用 Obfuscated Ticket Age - Ticket Age Add 计算出 Ticker 的有效期 Ticker Age，和数据库中取出的 Ticker 创建时间做比较。如果 Ticker Age 在 Ticket Lifetime 内，服务器则接受 0-RTT，反之拒绝 0-RTT。可以在计算的时候加上往返延迟，减少误差。

第二个措施是检查 Ticker 是否被使用。服务器在为每一个客户端发送的 NewSessionTicket 中的 Ticker 是由 Ticket Nonce 进过 HKDF 生成的，保证了不同客户端拥有不同的 Ticker，在服务器生成 Ticker 的同时，把 Ticker 作为 Key，1 作为 Value 保存到 Redis 数据库，保存所有未使用的 Ticker。当客户端使用 0-RTT 握手的时候，服务器通过的的 Binder 验证之后，还要去数据库检查 Ticker 是否存在，如果客户端发送的是未使用的 Ticker 一定能在 Redis 数据库中找到对应数据，然后接

受 0-RTT。并且删除 Redis 对应的 Ticker。如果重放 0-RTT，虽然能通过服务器的 Binder 检查，但是在查找 Redis 数据库时却返回为空，因此拒绝重放的 0-RTT，使用 ClientHello 中的 Key Share 参数进行 1-RTT 握手。

第三个措施，结合第一个和第二个措施做优化。服务器在生成 Ticker 保存到 Redis 数据库中的时候，设置 Redis 数据库中保存 Ticker 的超时时间，并且超时时间等于 Ticker Lifetime。当 Ticker 过期后，能及时自动删除数据库中过期的 Ticket，避免过多占用数据库。服务器接收到 0-RTT，首先检查 Ticker 是否过期，如果过期，则数据库会自动删除对应的 Ticker，因此查询 Ticker 失败，服务器可以拒绝此 0-RTT 请求，查询数据库不为空，则返回对应 Ticker 的创建时间，表明没有过期，进一步检查 Binder 是否正确。

第四个措施是服务器只接受幂等性的 0-RTT 请求。服务器收到 0-RTT 时，从 Resumption secret 导出 Early Data 的解密密钥，解密 Early Data 然后判断是否是非幂等性的请求，判断是否满足幂等性的请求的要求，如果不是，服务器将会拒绝 0-RTT，非幂等请求才能允许，比如是一个 GET 请求打开一个网站。这时使用 0-RTT 发送 GET 请求，可以网页可以更快展示出来，大多数的 GET 请求都是请求服务器上的静态资源，就算攻击者重放了 GET 请求的 0-RTT 数据包，最多只能使客户端发生页面的刷新，并不会服务器的对数据库进行操作和其重复请求操作。

第五个措施是：添加 Early Data 请求头，通知后端服务。Nginx 反向代理可以设置 proxy_set_header Early-Data $ssl_early_data; 由后端服务根据具体情况判断是否愿意接受这个 0-RTT 请求。如果拒接，后端服务拒绝 0-RTT 数据响应，并且发送 425 Too Early [rfc8470] 状态码。

第六个措施是客户端在 0-RTT 中发送自己认为是安全的请求，为了保证攻击者重放数据也得不到有效信息。非幂等请求通过 0-RTT 发送是不安全的，因为攻击者可以重放它们，但即使是幂等请求也可能不安全，有可能从返回信息中得到什么。具体应用程序可以控制，但浏览器一般认为通过 0-RTT 发送所有数据。在 Chrome 中，在握手确认之前仅发送 GET 请求。

通过共享数据库可以防止第二种攻击，以保证集群中的系统最多接受一次 0-RTT 数据，最多响应一次。使用 Ticker 共享，多个服务器使用同一个 Redis 数据库保存 Ticker，维护 Ticker 在分布式系统的的唯一性，群集中的服务器都将生成的 Ticker 保存到公共 Redis 数据库，统一维护 Ticker 唯一性和有效期。当集群中某一台服务器接收到 0-RTT 的时候，首先去查询数据库对应的 Ticker，如果是第一次发送并非重放攻击，并且此 Ticker 没有过期，下一步检查 Binder 的正确性。当攻

击者重放 0-RTT 的时候，负载均衡会将请求分发都没个后台服务，当其中一个请求查询 Ticker 时候，可以设置锁定数据库，防止重复读取合法的 Ticker。

第四章　测试与结果

前端使用 webpack 打包工具开发，运行在 9020 端口，而服务器使用 Node.js 开发，运行在 8600 端口，因为端口不同，浏览器的同源策略很阻止客户端和服务器的通信，需要配置代理，将 9020 端口数据转发到 8600 端口。同时为了使用 WebSocket 通信，服务器要配置启用 WebSocket 服务，访问地址不再是 http/https，而是 ws/wss。在浏览器/服务器模式下使用 Websocket 模拟 1-RTT 握手和 0-RTT 握手，打开网页首先进行 1-RTT 握手，客户端生成 32 字节的随机数、32 字节密钥，使用 x25519 曲线计算公钥，在 ClientHello 中发送给服务器，服务器收到 ClientHello 后也生成 32 字节随机数、32 字节私钥，同样使用 x25519 曲线计算公钥，并且使用客户端的公钥计算共享密钥，最后发送 ServerHello 给客户端。

客户端和服务器通过计算椭圆曲线私钥和公钥。完成协商参数和计算后使用 HKDF 计算导出密钥，计算 Ticker 和 Binder。客户端使用 0-RTT 握手时，使用对应的密钥加密 Early Data 并发送给服务器。服务器读取文件系统中的证书和证明密钥，证书通过 Certificate 消息发送给客户端，然后使用导出密钥计算 Certificate Verify 发送到客户端，最后使用私钥签署整个握手过程，发送 Finished 消息。客户端通过导出密钥计算 Finished 消息发送到服务器，到此完成密钥交换、服务器参数和认证。

服务器接收到客户端的 Finished 消息后，可以马上发送两次 NewSessionTicker 消息，用于之后的 0-RTT 握手。测试 0-RTT 时候，首先断开第一次的 WebSocket 连接，再次打开浏览器，使用 WebSocket 连接的时候，客户端首先检查缓存中是否有对应域名的 Ticket，如果有就从浏览器的 Localstorage 中读取对应的 Ticket，使用之前保存下来的复用密钥加密早期数据，连同 ClientHello 一起发送给服务器，服务器检查 Ticket，判断 Ticket 的有效性，选择接受早期数据还是拒绝早期数据。收到服务器 Finished 消息的客户端还需要发送 EndOfEarlyData 消息表明自己早期数据发送完成。

在客户端和服务器端使用 JavaScript 的 Buffer 操作，使用 WebSocket 发送消息时候，需要发送文本信息。为了方便观察，把参数、证书、导出密钥等显示在浏览器上。测试数据 Redis 初始化一万条数据量，使用 Node.js 作为后台服务器，测试抗重放功能。为此开发了具备功能齐全的后端服务器，比如注册、登陆、获取个人信息，获取最新新闻列表、转账等后台接口。

分别开发各种防御措施的中间件：1. 检查 Ticker 是否过期。2. 检查 Ticker 是

否被使用。3. 控制器判断请求是否带有 Early Data 的请求头，进一步判断是否为幂等性的请求。首先用户登录会在服务器生成 PSK 的 Ticker 和 Binder，其中 Ticker 有效期具体设置为 5 分钟，然后保存到数据库。测试在 Ticker 有效期内的重放攻击。使用中间件判断 Ticker，可以更灵活，重用性高。

对于 Ticker 有效期内的正常 0-RTT，验证通过后使用 HKDF 从 PSK 中导出密钥，解密 Early Data。服务器接收到过期的 Ticker 或者重放攻击的 Ticker，都会被拒绝，编写重放攻击程序，并发数量为 10 个，持续重放 60 秒。结果见图 1。添加的 0-RTT 请求头，后台接口可以这个请求头灵活判断怎么应答 0-RTT 消息。比如根据分时间段出来 0-RTT，多数攻击者活动在晚上，在早上或者服务器高访问量的时候允许通过 0-RTT，到了晚上服务器访问量少的时候可以拒绝 0-RTT。

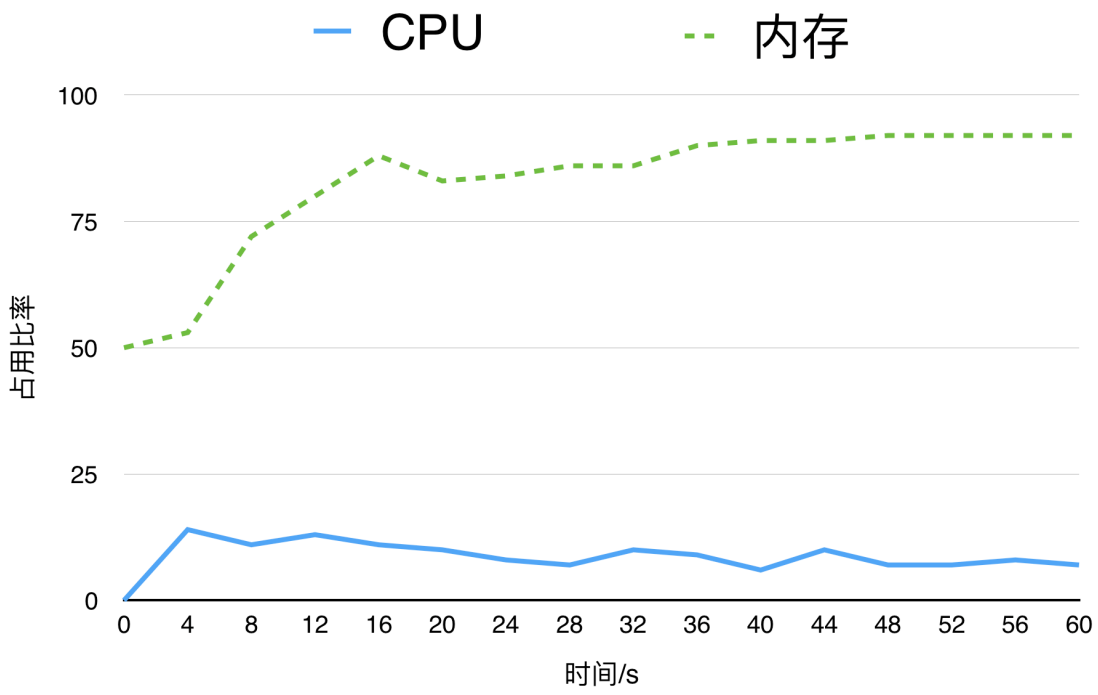


图 1: 服务器占用资源变化

第五章　总结与展望

5.1　总结

首次接触前端后端开发，在工具 webpack、vue.js、Node.js、expressjs 中遇到很多问题，上网搜索解决问题，看官网文档，慢慢从零基础到入门，勉强能完成基本功能。刚开始遇到浏览器跨域问题，虽然知道了它的原理，但是在解决方法上到处碰壁，在问答社区和请教同学才得以解决。实现过程中体会到 JavaScript 和 C/C++ 语言的很多差异，其中最突出的是 JavaScript 因为运行在 V8 引擎上，不能直接操作内存，在实现密钥生成、加密算法、密钥导出等操作的时候，需要用到 JavaScript 中 Buffer 操作，通过操作 Buffer 代替操作内存二进制数，很长一段时间没能适应，在编程中带来不少困难。

TLS 1.3 协议带来的启发：对称加密算法、MAC 算法、数字签名算法、密钥交换算法，四类基础算法的选择。对称加密算法选择更现代化、可证明安全的 AEAD 算法，其中最优的选择是 AES-256-GCM、ChaCha20-poly1305。数字签名和密钥交换可以选择速度更快，以及安全性相同但是密钥长度更短的椭圆曲线加密法。全面使用 SHA-2 算法，SHA-1 已经被破解了。全面使用 (EC)DH 密钥交换，考虑到前向保密性和使用 RSA 对服务器的性能要求，以及 RSA 私钥的保管有更大的风险。

少而精的 TLS 1.3，启用它可以更进一步改善用户体验，同时安全性也更有保障。对于 TLS 1.3 的 0-RTT 重放攻击，可以实现在 Ticker 有效期内的抗重放攻击，并且就测试结果来看，在重放攻击时服务器的占用资源也可以接受，在未确保 0-RTT 安全性时，服务器可以选择完全拒绝 0-RTT。措施一：检查 Ticker 有效期和唯一性一起使用，可以抵抗 0-RTT，但是会增加数据库的读写操作，减低服务器性能。措施二：检查 Ticker 有效期和添加 Early Data 请求头组合，根据实际业务需要灵活处理检查和处理 0-RTT。建议禁用 0-RTT 握手模式直到安全审计确定重放攻击不会对服务器造成威胁。测试 1-RTT 和 0-RTT 握手对比：1-RTT 模式下 TCP 握手开始到 TLS 完成握手发送第一个请求数据耗时 0.025504 毫秒，0-RTT 模式下 TCP 握手开始到 TLS 发送早期数据耗时 0.002199 毫秒，速度提升了 91.37%。

5.2 展望

本文实验测试比较简单，尽管实验中能解决 0-RTT 的重放攻击，可能存在效率更高效的方法没有发现，另外本文中 0-RTT 的前向保密性问题还没有涉及。有望能在以后的 TLS 版本更新中解决前向保密性问题。2015 年，Young Kyung Lee 和 Dong Hoon Lee [18] 等人提出从不可区分混淆中进行前向安全的非交互式密钥交换。2017 年 4 月,Günther, Felix and Hale [19] 等人提出利用一种可穿透的密钥封装方案，该方案允许每个密文只被解密一次解决前向保密性。2017 年 6 月，Hale, Britta and Jager [20] 等人提出可证明安全的简单安全模型。相信关于 TLS 1.3 的 0-RTT 问题能得到更好解决，为每个用户和每个设备提供更快，更安全的网络。

附录 A　HKDF

HKDF 第一步: 提取，对输入的初始密钥材料进行提取，输出固定长度的伪随机密钥 (PRK)

```
function hkdf_extract( hash, hash_len, ikm, salt ) {
    const b_ikm = Buffer.isBuffer( ikm ) ? ikm : Buffer.from( ikm );
    const b_salt = ( salt && salt.length ) ? Buffer.from( salt ) : Buffer.alloc( hash_len, 0 );

    return createHmac( hash, b_salt ).update( b_ikm ).digest();
};
```

HKDF 第二步: 拓展。将第一步的伪随机密钥做拓展，输出密钥材料 (OKM)

```
function hkdf_expand( hash, hash_len, prk, length, info ) {
    const b_info = Buffer.from( info || '' );
    const info_len = b_info.length;
    const steps = Math.ceil( length / hash_len );
    if ( steps > 0xFF ) {
        throw new Error( `OKM length ${length} is too long for ${hash} hash` );
    }
    // use single buffer with unnecessary create/copy/move operations
    const t = Buffer.alloc( hash_len * steps + info_len + 1 );
    // T = T(1) | T(2) | T(3) | ... | T(N)
    for ( let c = 1, start = 0, end = 0; c <= steps; ++c ) {
        // add info
        b_info.copy( t, end );
        // add counter
        t[ end + info_len ] = c;

        createHmac( hash, prk )
            // use view: T(C) = T(C-1) | info | C
            // 取T(C-1) t.slice( start, end + info_len + 1 )
            .update( t.slice( start, end + info_len + 1 ) )
            .digest()
            // put back to the same buffer
            // end 目标开始的位置
            // buf1.copy(buf2, 8); buf1 复制到 buf2 8 位置开始里
            .copy( t, end );
        start = end; // used for T(C-1) start
        end += hash_len; // used for T(C-1) end & overall end
    }
    return t.slice( 0, length );
};
```

完整的 HKDF 函数，进行一次提取和一次拓展

```
function hkdf( ikm, length, { salt='', info='', hash='SHA-256' } = {} ) {
    salt = nonce(length)
    hash = hash.toLowerCase().replace( '-', '' );
    // 0. Hash length
    const hash_len = 32
    // 1. extract
    const prk = hkdf_extract( hash, hash_len, ikm, salt );
    // 2. expand
    return hkdf_expand( hash, hash_len, prk, length, info );
}
```

附录 B　AEAD

AEAD 算法实现: ChaCha20 流加密算法。第一步用常量、密钥、计数器初始化矩阵。第二步初始矩阵置换，包括 10 轮行置换和 10 轮列置换，共 20 轮运算，其中 1 轮操作需要 4 次四分之一置换操作。第三步生成密钥流。变换后的矩阵和初始化矩阵相加，得到 512 位的密钥比特流。第三步加密。密钥比特流和明文异或

初始化矩阵

```
var Chacha20 = function(key, nonce, counter) {
  this.input = new Uint32Array(16);

  // https://tools.ietf.org/html/draft-irtf-cfrg-chacha20-poly1305-01#section-2.3
  this.input[0] = 1634760805;
  this.input[1] = 857760878;
  this.input[2] = 2036477234;
  this.input[3] = 1797285236;
  this.input[4] = U8TO32_LE(key, 0);
  this.input[5] = U8TO32_LE(key, 4);
  this.input[6] = U8TO32_LE(key, 8);
  this.input[7] = U8TO32_LE(key, 12);
  this.input[8] = U8TO32_LE(key, 16);
  this.input[9] = U8TO32_LE(key, 20);
  this.input[10] = U8TO32_LE(key, 24);
  this.input[11] = U8TO32_LE(key, 28);
  this.input[12] = counter;
  this.input[13] = U8TO32_LE(nonce, 0);
  this.input[14] = U8TO32_LE(nonce, 4);
  this.input[15] = U8TO32_LE(nonce, 8);
};
```

四分之一轮变换

```
Chacha20.prototype.quarterRound = function(x, a, b, c, d) {
  x[a] += x[b]; x[d] = ROTATE(x[d] ^ x[a], 16);
  x[c] += x[d]; x[b] = ROTATE(x[b] ^ x[c], 12);
  x[a] += x[b]; x[d] = ROTATE(x[d] ^ x[a], 8);
  x[c] += x[d]; x[b] = ROTATE(x[b] ^ x[c], 7);
};
```

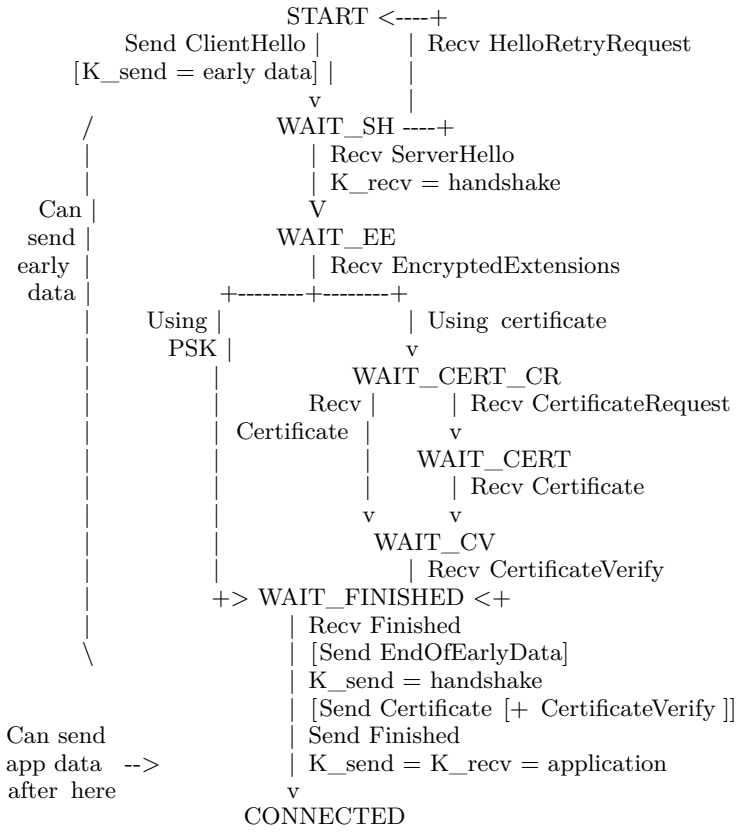
poly1305 算法: 输入 32 字节的密钥和任意长度的消息比特流，输入 16 字节的 MAC。将密钥分为 8 字节两部分，做预处理。poly1305 的密钥使用 ChaCha20 变换后的矩阵。

伪代码:

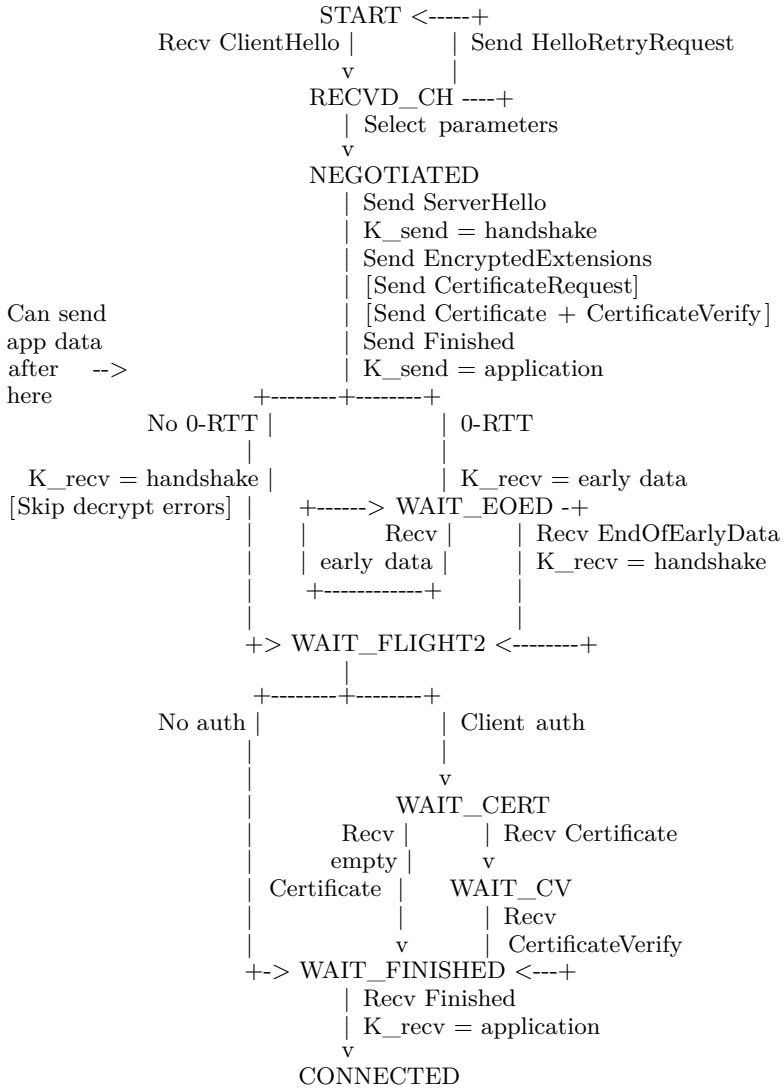
```
poly1305_key_gen(key,nonce):
  counter = 0
  block = chacha20_block(key,counter,nonce)
  return block[0..31]
  end
```

附录 C　TLS 1.3 状态机

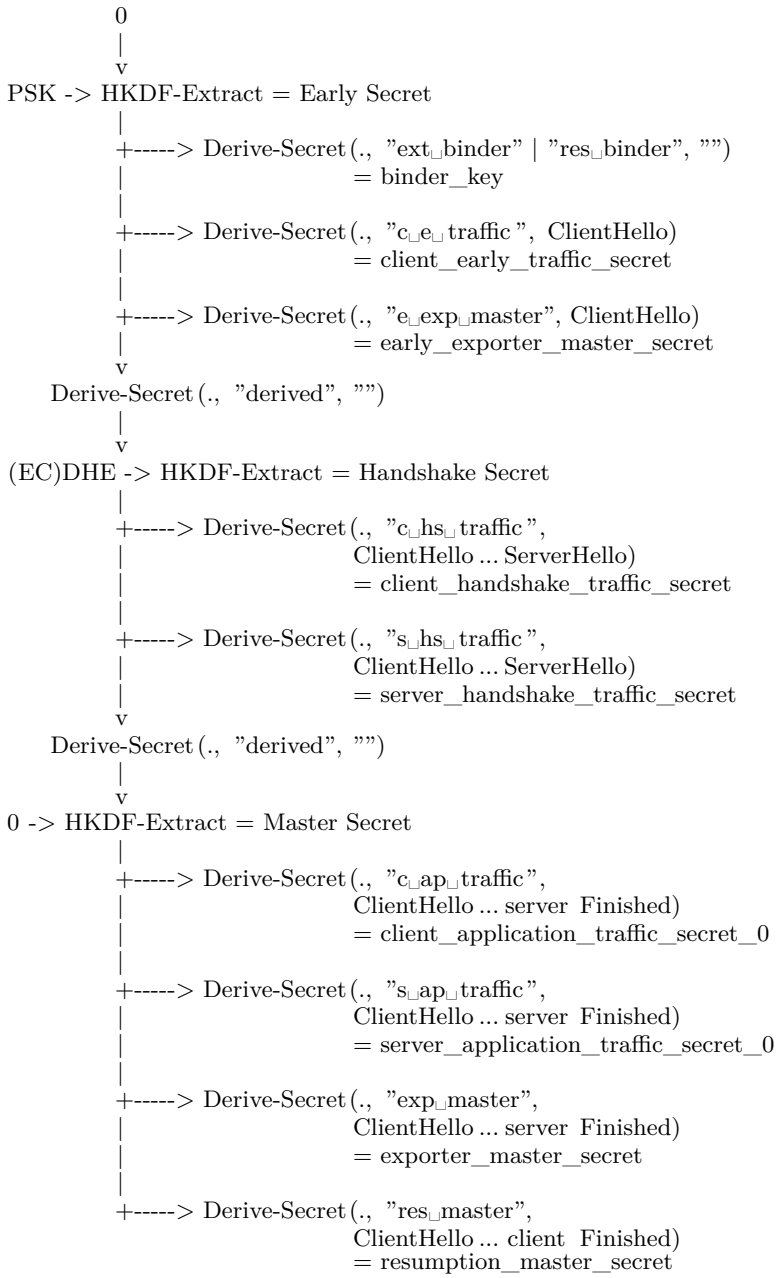
C.1　客户端状态机



C.2 服务器状态机


START <-----+
Recv ClientHello | | Send HelloRetryRequest
v |
RECVD_CH ----+
| Select parameters
v
NEGOTIATED
| Send ServerHello
| K_send = handshake
| Send EncryptedExtensions
| [Send CertificateRequest]
Can send | [Send Certificate + CertificateVerify]
app data | Send Finished
after --> | K_send = application
here +--------+--------+
No 0-RTT | | 0-RTT
| |
K_recv = handshake | | K_recv = early data
[Skip decrypt errors] | +------> WAIT_EOED -+
| | Recv | | Recv EndOfEarlyData
| | early data | | K_recv = handshake
| +------------+ |
| |
+> WAIT_FLIGHT2 <--------+
|
+--------+--------+
No auth | | Client auth
| |
| v
| WAIT_CERT
| Recv | | Recv Certificate
| empty | v
| Certificate | WAIT_CV
| | | Recv
| v | CertificateVerify
+-> WAIT_FINISHED <---+
| Recv Finished
| K_recv = application
v
CONNECTED

附录 D　密钥导出流程图



几点说明：

- HKDF-Extract 画在图上，它为从顶部获取 Salt 参数，从左侧获取 IKM 参数，它的输出是底部，和右侧输出的名称。
- Derive-Secret 的 Secret 参数由传入的箭头指示。例如，Early Secret 是生成 client_early_traffic_secret 的 Secret。
- ”0” 表示将 Hash.length 字节的字符串设置为零。

参考文献

[1] Sirohi P, Agarwal A, Tyagi S. A comprehensive study on security attacks on SSL/TLS protocol [C]. In 2016 2nd International Conference on Next Generation Computing Technologies (NGCT). Oct 2016: 893–898.

[2] Ivanov O, Ruzhentsev V, Oliynykov R. Comparison of Modern Network Attacks on TLS Protocol [C]. In 2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S T). Oct 2018: 565–570.

[3] Kyatam S, Alhayajneh A, Hayajneh T. Heartbleed attacks implementation and vulnerability [C]. In 2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT). May 2017: 1–6.

[4] Cremers C, Horvat M, Scott S, et al. Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication [C]. In 2016 IEEE Symposium on Security and Privacy (SP). May 2016: 470–485.

[5] Force I E T. The Transport Layer Security (TLS) Protocol Version 1.3 [OL]. https://tools.ietf.org/html/rfc8446/，2018-08/2019-04-04.

[6] Force I E T. An Interface and Algorithms for Authenticated Encryption [OL]. https://tools.ietf.org/html/rfc5166/，2008-03/2019-04-04.

[7] Lan X, Xu J, Zhang Z, et al. Investigating the Multi-Ciphersuite and Backwards-Compatibility Security of the Upcoming TLS 1.3 [J]. IEEE Transactions on Dependable and Secure Computing. 2019, 16 (2): 272–286.

[8] 张兴隆，程庆丰，马建峰. TLS 1.3 协议研究进展 [J]. 武汉大学学报 (理学版). 2018, 64(06): 471–484.

[9] 张兴隆，程庆丰，马建峰. 增强 TLS 1.3 中 Early data 安全性的协议 [J]. 网络与信息安全学报. 2017, 3(12): 8–16.

[10] WeMobileDev. 基于 TLS1.3 的微信安全通信协议 mmtls 介绍 [OL]. https://github.com/WeMobileDev/article/blob/master/基于 TLS1.3 的微信安全通信协议 mmtls 介绍.md，2017-06-04/2019-04-04.

[11] Yu Y, Xu M, Yang Y. When QUIC meets TCP: An experimental study [C]. In 2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC). Dec 2017: 1–8.

[12] Delignat-Lavaud A, Fournet C, Kohlweiss M, et al. Implementing and Proving the TLS 1.3 Record Layer [C]. In 2017 IEEE Symposium on Security and Privacy (SP). May 2017: 463–482.

[13] Force I E T. HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [OL]. https://tools.ietf.org/html/rfc5869/, 2010-05/2019-04-04.

[14] Velea R, Gurzău F, Mărgărit L, et al. Performance of parallel ChaCha20 stream cipher [C]. In 2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI). May 2016: 391–396.

[15] De Santis F, Schauer A, Sigl G. ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications [C]. In Design, Automation Test in Europe Conference Exhibition (DATE), 2017. March 2017: 692–697.

[16] Sun L, Luo Z, Wu Y, et al. A technique for preventing replay attack in road networks [C]. In 2012 7th International Conference on Computer Science Education (ICCSE). July 2012: 807–810.

[17] Fischlin M, Günther F. Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates [C]. In 2017 IEEE European Symposium on Security and Privacy (EuroS P). April 2017: 60–75.

[18] Lee Y K, Lee D H. Forward Secure Non-Interactive Key Exchange from Indistinguishability Obfuscation [C]. In 2015 5th International Conference on IT Convergence and Security (ICITCS). Aug 2015: 1–4.

[19] Günther F, Hale B, Jager T, et al. 0-RTT Key Exchange with Full Forward Secrecy [C]. 04 2017: 519–548.

[20] Hale B, Jager T, Lauer S, et al. Simple Security Definitions for and Constructions of 0-RTT Key Exchange [C]. 06 2017: 20–38.

致谢

感谢我的论文指导老师，(你的老师名字) 老师。(你的老师名字) 老师在我写论文的时候做出了指导性的意见，也是我大学学习的启蒙老师，让我体会到计算的美妙。在论文撰写过程中及时对我遇到的困难和疑惑给予教导，提出了很多重要的改进性的意见，在我写论文的过程里，投入了不少的时间和精力。对我耐心教导，为我指明方向，顺利完成论文撰写。

感谢四年以来教过我的每一位老师。深深的佩服他们的专业知识，从零开始教导我入门计算机科学，四年以来学习了很多，从最基本的计算机二进制到揭开网络连接神秘面纱的 TCP/IP 协议，每一样知识都深深触动了我，让我感受到计算的美妙。

感谢学校学院。大学学习的四年里，给我们提供了一个良好的学习环境和生活环境，有很多政策，帮助我们贫困生顺利完成四年学业，顺利进入社会。

感谢饭堂的叔叔阿姨。吃了四年饭堂，饭菜汤很便宜很好吃，谢谢你们每天一大早为我们准备最棒的早餐，工作认真负责任，让我吃了四年干净卫生的食物，从大一到大四，吃你们做的饭菜，我足足长了 10 多斤肉，每天都能吃好睡好学习好。

感谢五位与我朝夕相处了四年的舍友。时间过得很快，大一到大四，通过生活学习了四年，有汗水也有泪水，那些年那些日子，和他们一块生活学习，相互学习互相借鉴，这四年过得很愉快很充实，在我最困难的时候能推我一把，帮助我度过难关，包容我迁就我，很感谢他们的支持和鼓励，毕业之后，各奔东西，希望还是友情长存。

感谢我的父母。辛苦把我养大成人，一路上的支持和鼓励，给我精神上和金钱上的支持，完成这 20 多年的学习路途，从小学初中高中最后到大学，现在终于要结束了。

感谢西三 404 宿舍。夏天里，空调带给我清凉甜蜜，让夏天时刻保持惊喜，酷暑里解救生命，没有空调怕是要热成咸鱼。我没能忍受热浪离开宿舍到处浪，宿舍学习天天向上，空调 Wifi 最给力。感谢五分钱一次的热水澡，冰冷的季节温暖我的心，洗掉一天的疲劳，那温暖的热水，很有感觉，幸福了这个季节。五分钱你买不到一杯奶茶买不到一个汉堡满足你个胃口，但在西三，没有烦恼是一次热水澡不能解决的，如果有，那就两次。

最后感谢自己。献给曾经在人生道路上迷茫的我，唯有灵魂和梦想不能辜负。