

美图HTTPS优化探索与实践



原创 2017-07-05 李子昂 高可用架构

HTTPS 是互联网安全的基础之一，然而引入 HTTPS 却会带来性能上的损耗。本文作者深入解析了 HTTPS 协议优化的各个方面，对实战很有帮助。

2012 年斯诺登（Edward Snowden）爆出棱镜门事件后，互联网安全问题日益得到大家的重视。去年 Apple 宣布 2017 年 1 月 1 日之前实现所有的 App 能够安全地接入服务器，这项声明来自于 iOS9 时代的应用程序安全传输功能（App Transport Security）。逾期没有采用 HTTPS 的 app 将无法通过审核并遭到下架。同年美图也在 11 月完成了全网的 HTTPS 改造，将服务的安全级别到了一个新的高度。

本文将科普式的介绍 HTTPS 协议以及美图在 HTTPS 优化方面的探索与实践。

Abstract

限于篇幅，本文不对 TLS/SSL 协议的安全性做过多的假设与讨论，将围绕 HTTPS 的体验（速度）优化介绍一下几个方面：

- TLS/SSL 协议浅析
- HTTPS 优化探索
- HTTPS 优化实践

SSL：安全套接字层

概述

听到 SSL 协议很多同学可能立刻会想起一个问题：SSL 协议和 TLS 协议到底有什么区别呢？

HTTPS 是基于 **SSL** 的加密传输协议，使用 **SSL** 来协商加密密钥。SSL 协议最早是由网景公司（Netscape）开发，但是随着网景的没落，现在由 IETF 负责维护，最初的版本也已经重新冠名 **TLS**（安全传输层协议）1.0（1999 年）。因此现在大部分协议是基于 TLS 的，尽管是相似的东西。所以这两个协议其实是同一个东西，叫什么名字都可以。

为了方便记录，本文后续将使用 **SSL** 协议来代指 **SSL/TLS** 协议。

截止目前 **SSL/TLS** 协议族中有 7 种协议：

SSL v1, SSL v2, SSL v3, TLS v1.0, TLS v1.1, TLS v1.2, TLS v1.3(draft)：

- SSL v1 从未正式公开。
- SSL v2 协议设计有缺陷，不安全。
- SSL v3 老旧过时，缺乏一些新的密钥特性。
- TLS v1.0 在很大程度上是安全的，至少没有曝光重大的安全漏洞。
- TLS v1.1 和 TLS v1.2 目前都没有著名的安全漏洞曝光。

- TLS v1.3 仍然在草案阶段，而且有待时间检验。

常用 Linux 发行版默认 OpenSSL 版本：



操作系统	内置OpenSSL版本
Centos 5	0.9.8e
Centos 6	1.0.1e
Centos 7	1.0.1e
Ubuntu 14.04 LTS	1.0.1f
Ubuntu 16.04 LTS	1.0.2g
Debian 7 Wheezy	1.0.1e
Debian 8 Jessie	1.0.1k

高可用架构

协议栈关系

我们先来看看 SSL 协议到底工作在哪里。

Layer		Protocol data unit (PDU)
Host layers	7. Application	Data
	6. Presentation	
	5. Session	
	4. Transport	Segment (TCP) / Datagram (UDP)
Media layers	3. Network	Packet
	2. Data link	Frame
	1. Physical	Bit

高可用架构

TCP 7 层协议栈是我们都熟悉的内容，SSL 协议是工作在 **表示层** 的协议。这也就是说 SSL 需 TCP/UDP 之上工作；在 HTTP/FTP/SNMP 等协议之前创建会话。顾名思义 HTTPS 就是基于 SSL 协议的加密的 HTTP协议。



TCP/IP protocols
NNTP · SIP · SSI · DNS · FTP · Gopher · HTTP · NFS · NTP · DHCP · SMPP · SMTP · SNMP · Telnet · LDAP · SMB · BGP · FCIP
MIME · SSL · TLS · XDR
Sockets (session establishment in TCP / RTP / PPTP)
TCP · UDP · SCTP · DCCP
IP · IPsec · ICMP · IGMP · OSPF · RIP
PPP · SBTv · SLIP

SSL 协议栈内容

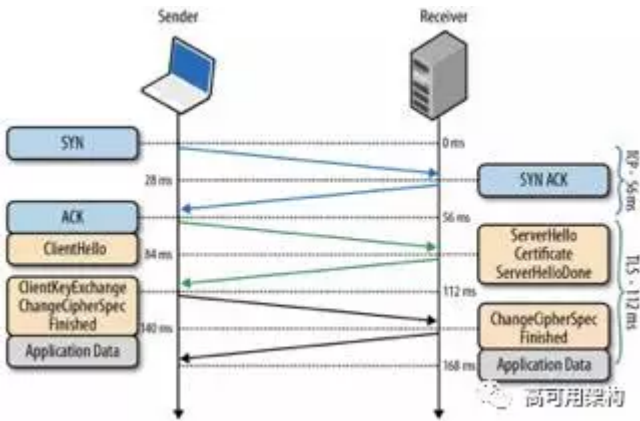
整个 SSL 协议栈包含了 **三种** 类型的协议：

- 握手协议：用于协商 SSL 密钥
- 记录协议：用于记录 SSL 会话相关信息
- 警报协议：用于通知对端停止 SSL 会话

这里我们结合抓包来重点看一下 **握手协议** 的工作过程。

SSL：握手过程

先来一张图，看一看 SSL 握手的过程：



图上蓝色部分是 TCP 的握手，灰色部分是应用层加密的数据传输（HTTPS 数据）。可以看到，整个握手过程设计，server 与 client 交互的过程大致有以下几个部分：



- client hello
- server hello
- client key exchange
- change cipher spec

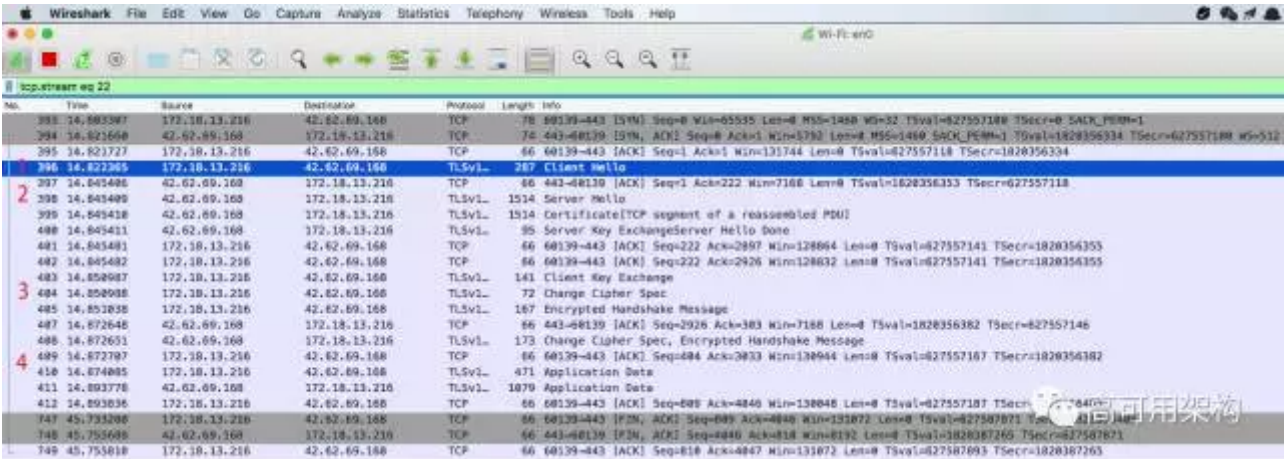
概括起来，前两个过程 **生成** 了非对称加密的重要参数，创建了非对称密钥。后两个过程 **交换** 了非对称加密的公钥。

欲详细了解握手的过程是如何发生的，交换了哪些内容，请参考我的另一篇博客：

[从 TCP 建立连接到 HTTPS 接收到第一个数据包，到底发生了什么？\[1\]](#)

接下来让我们通过简单的抓包我们可以看到整个 SSL 协议握手与创建会话的全过程。

`Wireshark` 是我最喜欢的抓包工具之一，另一个是 `tcpdump`。为了生动形象更易阅读，我们用 `Wireshark` 来抓包了解整个过程。



1. client hello (client)
2. server hello + certificate exchange (server)
3. change cipher spec + finished (client)
4. change cipher spec + finished (server)

通过抓包，往往我们可以清楚地看到从建立 TCP 连接到 TCP 连接断开，整个 SSL 握手的过程。这个过程是艰辛复杂的，从握手开始到真正的加密数据发送，之间有多个 RoundTrip。因此我们的优化过程就是从这里开始入手。

SSL 流量优化

顺着这样一个思路，先来看下优化到底能从哪些地方入手：

- 优化 SSL 握手之前过程
 - TCP 优化
- 优化 SSL 握手过程
 - False start
 - 优化握手流程
 - 加快密钥计算

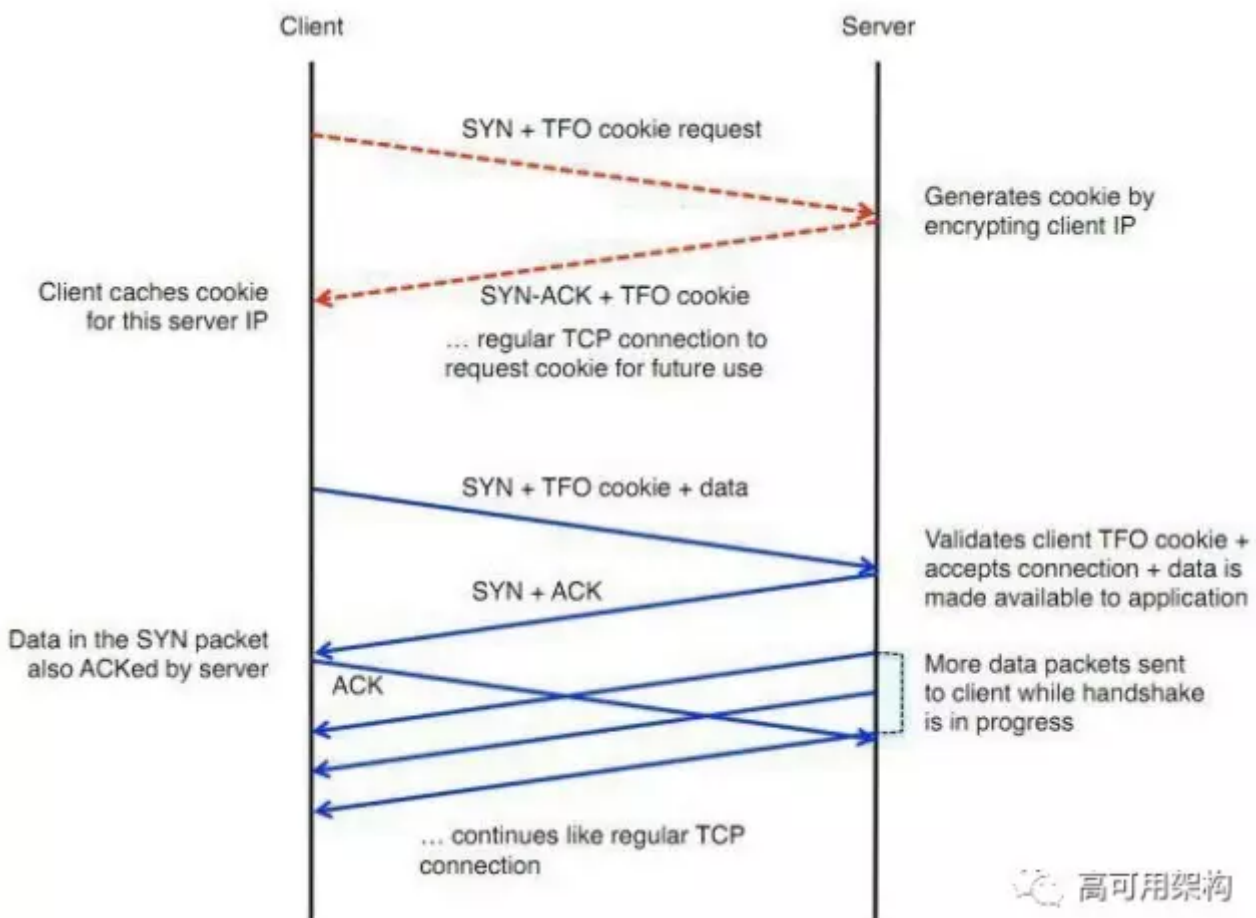


- 优化 SSL 握手之后过程
 - HTTP/2 || SPDY
 - 内容压缩
 - 对称加密套件选择

优化 SSL 握手前的过程

TCP Fast Open

TCP Fast Open(以下简称 TFO)目的在于简化 TCP 握手的过程，通过一定的协商过程（SYN 携带 cookie 信息）使得下一次握手的时候在 SYN 包中就可以携带数据，同时 Server 可以在发出 SYN ACK 之后立即开始发送数据。因此如果我们的 SSL 握手建立 TCP 连接的时候能够启用 TFO，那么我们的 SSL 握手流量就可以减少 **至少一个 RTT**。



但是，由于 Server linux 内核过于低，有的内核并没有支持 TFO。这点在我编译 nginx 的时候就遇到了服务端内核过低的问题。

TCP_FASTOPEN 特性在 kernel-3.6 被客户端支持，在 kernel-3.7 被服务端支持。

想要开启 TFO，先要检查下你的内核是否支持。

TCP 参数调优

大家都知道 TCP 的几个重要的算法，这里要说的就是 TCP 的慢启动（Slow-Start）算法。

由于存在慢启动的特征，开始时的 TCP 窗口可能较为小，因此如果我们的 SSL 握手包非常的大，那么潜在的可能就是发生 TCP 分片，相当于增加了一个额外的 RTT。因此，如果能在不影响其他服务的情况下，调整窗口的大小，也是一种优化的思路。



关于一点，我们在后面还会继续说明。

优化 SSL 握手的过程

优化 SSL 握手过程是整个过程的核心，这里也是重点优化的方向。我们有以下几个思考优化的点：

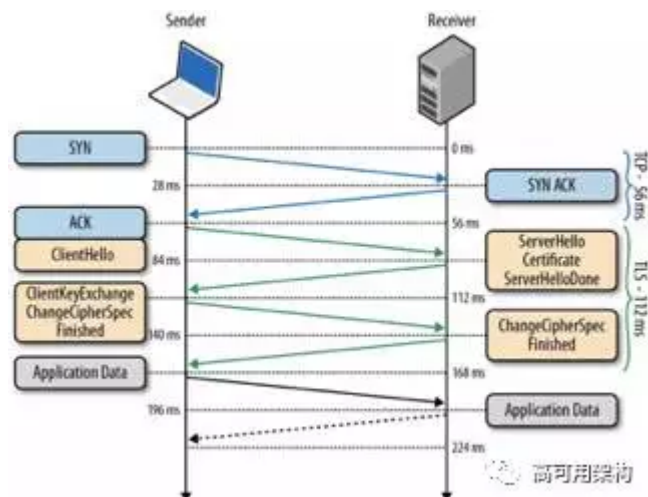
- TLS False Start
- HSTS
- Session Resumption
- OCSP Stapling
- 证书优化
- 非对称加密运算加速

SSL False Start

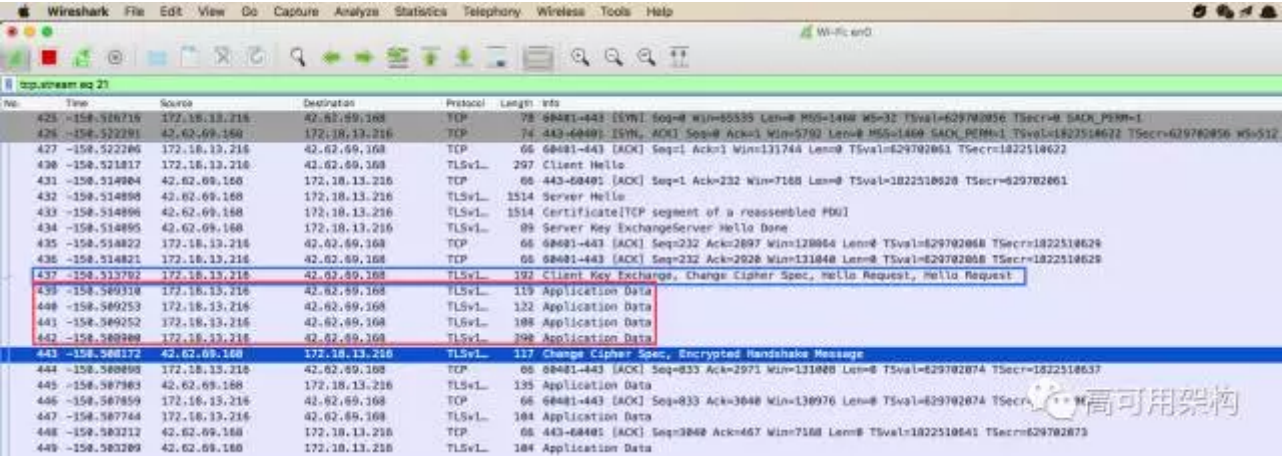
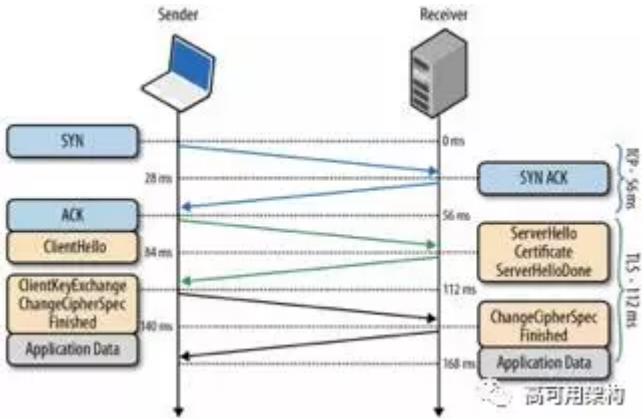
在 SSL 握手完成之前率先开始加密的 application 数据交互。这种思想很像刚刚提及的TFO思路。通过节省掉一次交互过程从而节省 RTT。实现的基础是 client 拿到server random 之后（server hello）其实就已经有了生成非对称加密所需要的密钥的 **3 个关键**：

server random ， client random ， pre-master-secret 。

未开启 False Start



开启了 False Start



通过抓包可以发现，在刚收到 437 号包协商还没结束的时候，应用层的数据已经开始了传输（439-442 号包），而到 443 号包才完成了握手。

在现代的 SSL False Start 实现中，主要有两种方式：

- NPN：Next Protocol Negotiation 协议文档[2]
NPN 最早引入 false start 思想，是 SSL 的扩展部分。不过 Chrome 51 中移除将会 NPN，因此应该尽快支持 ALPN 来替代
- ALPN Application Layer Protocol Negotiation
ALPN 是 Google 根据 HTTP2 设计的，计划替代 npn 的新协议，目的与 npn 相同，通过协商，优化 SSL 握手的过程。然而，ALPN 需要 OpenSSL 1.0.2 支持，当前主流服务器操作系统基本都没有内置这个版本。所以需要进行升级。

这两者本身设计用于协议协商的，因此也可以用于支持 HTTP2。但是他们之间有一些差异：

- NPN 是服务端发送所支持的 HTTP 协议列表，由客户端选择
- NPN 的协商结果是在 Change Cipher Spec 之后加密发送给服务端
- ALPN 是客户端发送所支持的 HTTP 协议列表，由服务端选择
- ALPN 的协商结果是通过 Server Hello 明文发给客户端

这两个协议用于 SSL False Start 本身就是使用了副作用。不过，为了开启 False Start，加密套件必须支持前向保密，例如 ECDHE_RSA 加密套件。

HTTP Strict Transport Security

HTTP Strict Transport Security (简称HSTS) 是一个安全功能，告诉浏览器只能通过 HTTPS 访问当前资源，禁止 HTTP 方式。最初目的防范降级攻击 (Downgrade attack) 或中间人攻击 (Man-in-the-middle attack)



我们肯定遇到过这样的情况：访问一个 HTTP 开头的网站，然后浏览器自动帮我们跳转到了HTTPS 页面。这意味着一次潜在的重定向。

HSTS 通过内置的 preload list 保存了一份可以定期更新的列表，对于列表中的域名，即使用户之前没有访问过，也会使用 HTTPS 协议。如果你的服务端开启了 HSTS 这个选项，那么客户端可以直接访问 HTTPS 页面，从而避免了一次无谓的跳转，其实也是加速了访问的过程。

这样做的好处是带来安全性的同时，避免了潜在的重定向带来的一次额外 RTT。

然而全站 HTTPS + HSTS，导致一旦更换回 HTTP，处于 list 的老用户会被无限重定向。

chrome 查看本地 hsts preload list 的方式是访问 `chrome://net-internals/#hsts`

Query domain

Input a domain name to query the current HSTS set:

Domain:

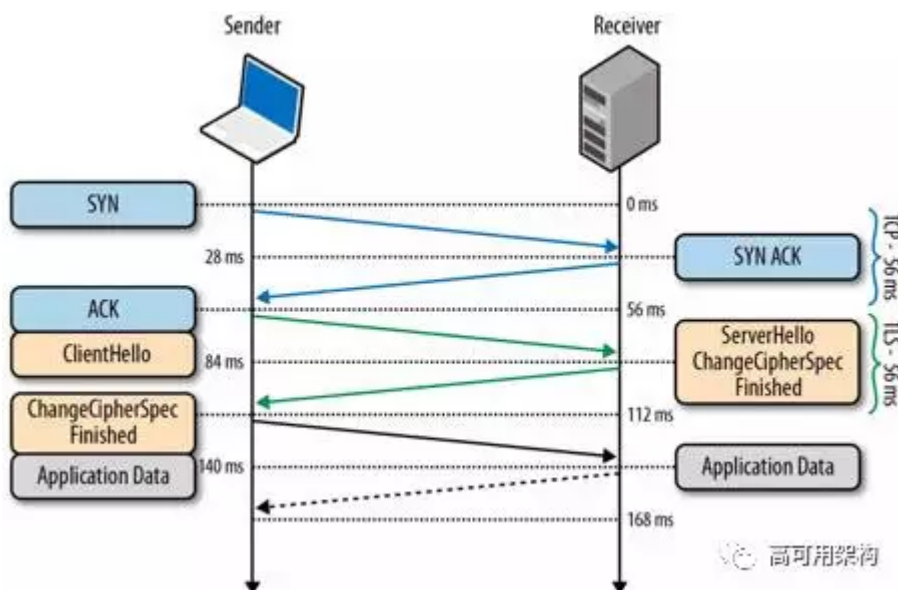
Found:

```
static_sts_domain:
static_upgrade_mode: unknown
static_sts_include_subdomains:
static_sts_observed:
static_pkp_domain:
static_pkp_include_subdomains:
static_pkp_observed:
static_spki_hashes:
dynamic_sts_domain: varycloud.com
dynamic_upgrade_mode: strict
dynamic_sts_include_subdomains: false
dynamic_sts_observed: 1496322347.776217
dynamic_pkp_domain:
dynamic_pkp_include_subdomains:
dynamic_pkp_observed:
dynamic_spki_hashes:
```

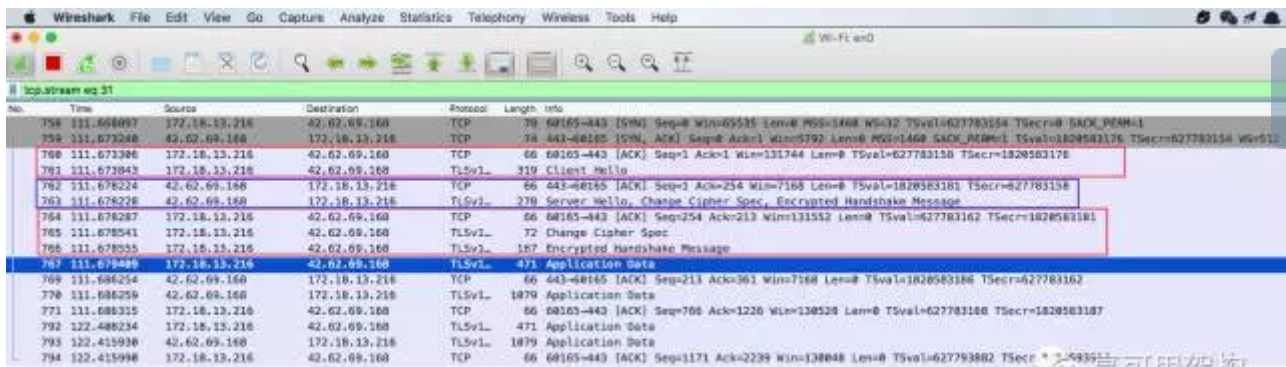
高可用架构

SSL Session Resumption

这里使我们优化大点的重点。Session Resumption意为会话恢复。顾名思义，这里的会话正是指恢复先前的 SSL 会话。通过会话恢复技术我们可以以最小的代价，规避最耗时的握手过程。收益可观。



我们先上抓包:



通过抓包可以看到，开启了 Session Resumption 的 SSL 握手，在 2 个 RTT 的时候就已经完成了握手。这是如何做到的呢？

我们先来看3个名词概念：

- Session ID：会话 ID，用于唯一标识一个 SSL 会话。
- Session Ticket：会话凭据，记录了本次会话相关的密钥与认证信息。
- Session Cache：会话缓存，用于缓存会话相关的信息。

目前实现会话恢复大致有 2 种方式：Session ID 恢复会话和 Session Ticket 恢复会话。

Session ID 方式

TLS 握手中生成的 Session ID，服务端可以将 Session ID 和 协商后的信息对应存起来。同时，浏览器也可以保存 Session ID，并在后续的 ClientHello 握手中带上它，如果服务端能找到与之匹配的信息，就可以完成一次快速握手。

```
▼ Handshake Protocol: Server Hello
  Handshake Type: Server Hello (2)
  Length: 100
  Version: TLS 1.2 (0x0303)
  ▶ Random
  Session ID Length: 32
  Session ID: d8e92ca73a9806672617095cb3d41a9c3e5f726f712d8de3...
  Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
  Compression Method: null (0)
  Extensions Length: 28
```

这样做带来的弊端也很明显：

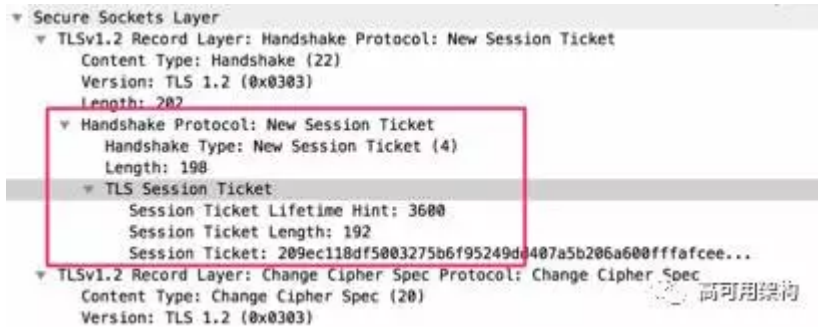
- 由于目前大多是大型互联网结构。负载均衡中，多机之间往往没有同步 Session 信息，如果客户端两次请求没有落在同一台机器上就无法找到匹配的信息，导致失败。
- 服务端存储 Session ID 对应的信息不好控制失效时间，太短起不到作用，太长又占用服务端大量资源。

为了解决这个问题，可以通过 IP hash 来负载，但是依然容易因为网络环境变化导致握手的失败。通过 Session cache 共享也是一种解决方案，nginx 提供了本机的共享，但是跨节点的共享就需要通过一个中心化的 Session cache 来实现。这无疑增加了跟多的成本。

Session Ticket 方式

Session Ticket 是通过服务端安全密钥加密过的会话信息，最终保存在 client。浏览器如果在 ClientHello 时带上了 Session Ticket，只要服务器能成功解密就可以完成快速握手。这种方式缓解了 Session ID 带来的潜在的握手失败的风险。

然而，Session Ticket 的加密使得 **必须保证所有 server 上的加密密钥相同** 否则负载均衡到不同机器的时候因为无法解密一样会导致失败。



看似，Session Ticket 更容易解决 Session id 的一些痛点。然而在我们的抓包中，很容易发现，因为 Session Ticket 信息过大，导致 TCP 分片。特别是在开始阶段门限值小，更容易出现这个问题。

OCSP Stapling

OCSP stapling 是 OCSP 的一个扩展协议，那么什么是 OCSP 呢？

在 SSL 握手阶段，客户端验证证书有效状态通常有两个方式：

- **CRL** (Certificate Revocation List, 证书撤销名单)
- **OCSP** (Online Certificate Status Protocol, 在线证书状态协议)

CRL 时效性差，而且 list 会越来越大，浏览器更新通常不及时（以天为单位）。

OCSP 是为了解决旧的证书检查协议的弊端的实时协议，实时解析证书的有效性。

然而，某些客户端会在 SSL 握手阶段进一步协商时，实时查询 OCSP 接口，并在获得结果前阻塞后续流程，这对性能影响很大。OCSP stapling 正是为了解决这些问题：将对于证书的验证过程代理到 server 上，减少 client 的压力。查询后打包下发 client。同时 OCSP stapling 支持 Certificate Transparency，证书安全性有保障。

通过 OpenSSL 内置工具可以查询服务端 OCSP Stapling 的状态：

```
openssl s_client -connect varycloud.com:443 -servername varycloud.com -status -tlsextdebug < /dev/null 2
OCSP response:
OCSP Response Data:
  OCSP Response Status: successful (0x0)
  Response Type: Basic OCSP Response
```

同一个请求，第一次可能没有开启 OCSP Stapling，原因在于 server 会先去获取 OCSP 的状态，**这需要**一个过程。



证书优化

证书优化是指减小不必要的握手开销，这里我们先不讨论加密运算的消耗。通常我们有一个证书的原则，证书包含中间证书但是不包含根证书，这是最小化的证书配置，因为根证书浏览器一般都会内置。而中间证书如果不一同下发，那么浏览器会递归的查询请求中间证书，这也是无意义的消耗。

通常来说，我们常用的非对称加密算法是 RSA。可以说，RSA 也是我们信息安全的基石。然而 RSA 的密钥长度是非常巨大的，计算量也是惊人的。

一种更新的解决方案是使用 DH 加密。这也是一种非对称加密。对于 DH 加密，常常与椭圆曲线一起使用被称为 ECDH。通过椭圆曲线优化之后的 DH 加密有更好的性能同时占用更小的密钥长度。

对称加密 Key 长度	RSA Key 长度	ECC Key 长度
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

高可用架构

ECDH 算法使用较短的密钥即可达到相同程度的安全性，这是因为它依赖椭圆曲线而不是对数曲线。ECC 是建立在基于椭圆曲线的离散对数问题上的密码体制，给定椭圆曲线上的一个点 P ，一个整数 k ，求解 $Q = kP$ 很容易；给定一个点 P 、 Q ，知道 $Q = kP$ ，求整数 k 确是一个难题。

OpenSSL 1.0.2 采用了 Intel 最新的优化成果。这个优化特性只是在 1.0.1L 之后才加入的，下图表格中 OpenSSL 的 1.0.1e 版本 `ecdh(nistp256)` 的性能只有 2548，而 OpenSSL 的 1.1.0b 版本 `ecdh(nistp256)` 性能能达到 10271，提高了 4 倍。

非对称加密运算加速

RSA 和 ECDHERSA 为什么会消耗 CPU 资源？

RSA 主要是对客户端发回来 `pre_master_secret` 进行解密，它消耗 CPU 资源的过程是私钥解密的计算；而 ECDHERSA 则有两个步骤：

1. 生成 ECC 椭圆曲线的公钥和几个重要的参数；
2. 对这几个参数进行签名，客户端要确保参数是服务端发过来的，就是通过 RSA 的签名来保证。

RSA 签名为什么消耗 CPU 呢？RSA 签名同样有两个步骤：

1. 首先它通过 SHA1 进行 Hash 计算；
2. 对 Hash 结果进行私钥加密，也就是最终消耗 CPU 的过程是私钥解密和私钥加密的计算。这两个计算为什么消耗 CPU？看下图公式，如果 e 或者 d 这个指数是一个接近 2 的 2048 次方的天文数字，那就非常消耗 CPU。这也就是 RSA 算法为什么消耗 CPU 的最直接的数学解释。

对于非对称加密的优化，常见的方式有

- 硬件加速卡
- GPU 加速集群代理计算

专门的硬件优化可以节约大量的时间，这点相信了解计算机原理的同学都能明白。同样 GPU 代理计算是将计算与握手分离，同样是专用集群专门加速，因此效率更高。

因为对这些具体的硬件没有专门的测试和了解，就不过多的讨论。

优化 SSL 握手之后的流程

在通过了握手之后，我们的流量进入了安全的通信通道。此时还有可以优化的空间么？

- 对称加密算法优化
- HTTP2 / SPDY（TCP 多路复用）

对称加密算法优化

经过了 SSL 握手之后的流量就进入了加密的通信通道，因此对称加密算法的效率其实也是十分重要的。然而因为大多数的对称加密算法并没有很大的优化空间，这里的优化往往得不到足够的重视。我们这里以一种特殊的加密算法为例进行介绍。

ChaCha20-Poly1305 是 Google 所采用的一种新式加密算法，性能强大，在 CPU 为精简指令集的 ARM 平台上尤为显著（ARM v8前效果较明显），在同等配置的手机中表现是 AES 的 4 倍。可减少加密解密所产生的数据量进而可以改善用户体验，减少等待时间，节省电池寿命等。

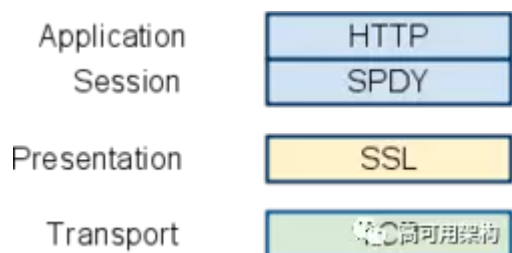
ARM v8 之后加入了 AES 指令。所以在这些平台上的设备，AES 方式反而比 chacha20 方式更快，性能更好。因此作为一个可选的方案，chacha20 可能对于一些老旧的机型（向嵌入式大牛咨询过之后了解到，目前来说还有很多在使用 arm v7 的硬件）来说更具优势。

SPDY || HTTP/2

对于流量的优化还有一种思路，那就是 TCP\SSL 链接的复用。讲到http的多路复用，就绕不开 HTTP/2。这次，我们从HTTP/2的始祖 SPDY 说起。

传统的http请求模型类似于 1:1 形式，单个 TCP（SSL）链接服务一个 HTTP 请求，很多情况下，TCP 链接资源并没有得到充分的利用。HTTP 多路复用能够有效地提高 TCP 链路的效率。

SPDY 是 Google 开发的基于 TCP 的传输层协议，用以最小化网络延迟，提升网络速度，优化用户的网络使用体验。IETF 对谷歌提出的 SPDY 协议进行了标准化，于 2015 年 5 推出了类似于 SPDY 协议的 HTTP 2.0 协议标准（简称HTTP/2）。谷歌因此宣布放弃对 SPDY 协议的支持，转而支持 HTTP/2。



简而言之，无论是 SPDY 还是 HTTP/2 目的都在于提高链接的效率。而且，HTTP/2 天然支持 SSL false start 和 HTTPS，安全性和效率上都提供了保障。

HTTPS 优化实践

这里我们将通过数据的对比来检验 SSL 优化的效果。我们从以下 3 个测试结果来分析 SSL 优化的效果。

- SSL Session Resumption
- OCSP stapling
- 加密算法优化与分析

测试方案

采用压力测试的方式，对比不同配置下，相同压力服务端的服务能力和延迟表现。

服务端配置

- CPU Info E5 6 核心 12 线程（超线程）
- 32G RAM
- Intel Corporation 82574L Gigabit Network 千兆网卡

测试 Server版本

- Linux kernel
 - 2.6.32-573.el6.x86_64
- nginx
 - version: nginx/1.10.2
- OpenSSL
 - OpenSSL 1.0.2l
- 压测工具
 - wrk 是一个专门用于测试 HTTP 请求响应的工具。功能强大，但是对于 HTTPS 支持不够友好。
 - wrk-go 是我使用 go 和自己的测试框架 perfm 重新实现的一个测试工具，专门增加了 HTTPS 相关的选项，方便测试。

▪ 文后有 wrk-go 链接，欢迎大家一起完善它！（这不是广告 ^ ^）[3]



SSL Session Resumption

测试结果

未开启会话恢复

```
./wrk -c 24 -t 24 -d 3m -H "Connection: Close" https://varycloud.com
Running 3m test @ https://varycloud.com
24 threads and 24 connections
Thread Stats      Avg          Stdev         Max      +/- Stdev
  Latency        4.76ms       8.34ms    174.25ms    97.01%
  Req/Sec        51.91        10.48      90.00       69.50%
223479 requests in 3.00m, 67.99MB read
Non-2xx or 3xx responses: 223479
Requests/sec:    1240.95
Transfer/sec:     386.58KB
```

开启会话恢复

```
./wrk -c 24 -t 24 -d 3m -H "Connection: Close" https://varycloud.com
Running 3m test @ https://varycloud.com
24 threads and 24 connections
Thread Stats      Avg          Stdev         Max      +/- Stdev
  Latency        1.71ms       10.17ms    165.85ms    98.78%
  Req/Sec       360.06        53.23     545.00      82.35%
1544216 requests in 3.00m, 469.78MB read
Non-2xx or 3xx responses: 1544216
Requests/sec:    8574.25
Transfer/sec:     2.61MB
```

通过对比可以看到，SSL 会话恢复如果能够被成功复用，那么提升非常可观。相比基准数据能够提升大约 **64%**。






同时由于略过了最消耗 CPU 的运算过程，可以看到系统的负载明显降低，同时 IO 明显提升，这说明服务能力得到了提高。

综上可以得出，`Session Resumption` 效果是非常明显的。

OCSP Stapling 测试分析

OCSP Stapling 是我们感兴趣的另一个特性。下面给出开启 OCSP Stapling 的表现测试数据。具体统计了平均耗时 `avg`，耗时差异 `diff`，数据包大小 `Data Length`。

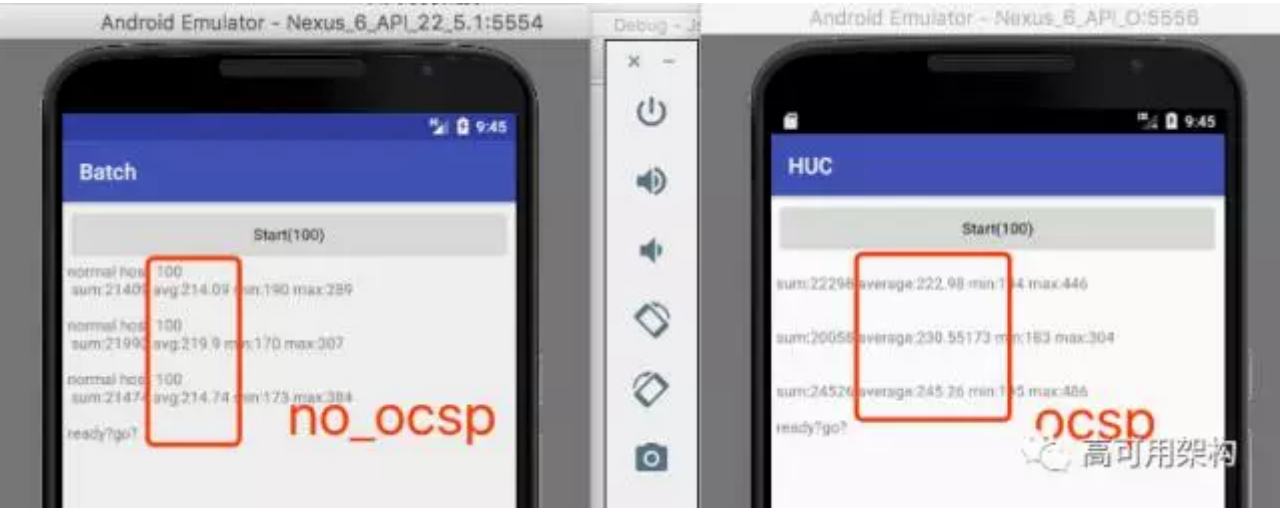
Time:ms		Data Length:bytes													
	TCP1	TCP2	TCP3	Client Hello	Server Hello	Certificate	Certificate Status	Server Key Exchange	Server Hello Done	Client Key Exchange	Client Cipher	Client Cipher	App Data	App Data	All
Server : 42.62.69.169 (OCSP Stapling)															
avg	0.000	57.286	0.101	2.353	61.427	0.850	0.026	←	←	16.858	←	56.873	2.379	57.430	255.536
Server : 42.62.69.168															
avg	0.000	56.021	0.098	2.338	60.747	0.365	x.xxx	0.114	←	11.488	←	55.394	2.489	56.074	245.189
diff	0.000	1.265	0.003	0.015	0.680	0.485	0.026	-0.114	x.xxx	5.370	x.xxx	1.479	-0.110	1.356	10.347
Data Length	78	74	66	274	1514	1514	619	←	←	192	←	117			

高可用架构

上图中白框圈中的部分为开启 OCSP 后握手步骤的差别及耗时相差最多的阶段。<---- 表示当前握手步骤和箭头指向的握手步骤是在同一个数据包。

分析

然而测试结十分令我们困惑：



理论上讲，OCSP Sapling 规避了一次客户端与服务端之间的通信，时间消耗上应该会有提升。然而却出现了下降。通过分析我们发现了端倪。

Client 与两者的对比，在经历各自 2000 请求，发现与开启 OCSP Stapling 的 Server 通信，统计上多消耗了 10.347ms 左右，最大的差异是 Client 收到 Certificate 后再到 Client Key Exchange，开启 OCSP Stapling 的 Server 的通信多消耗了 5.370ms。

对比两者握手差异，对应任务多了两部分

- 1. 新增加传输 Certificate Status 的数据包，包大小为 619 bytes。
- 2. 验证 Certificate Status 的合法性

由于网络传输耗时与所传输的数据量多少成正比，以到 server ping 的耗时为标准时间的话，从时间上推断传输 Certificate 加上 Certificate Status 耗时大概约为 16.1848

11.488 * (1514 + 619) / 1514 = 16.1848

多出来的大概 0.673ms 可认为是验证 Certificate Status 合法性的时间（如果不考虑两台服务器在通信时微小的网络差异的话）。



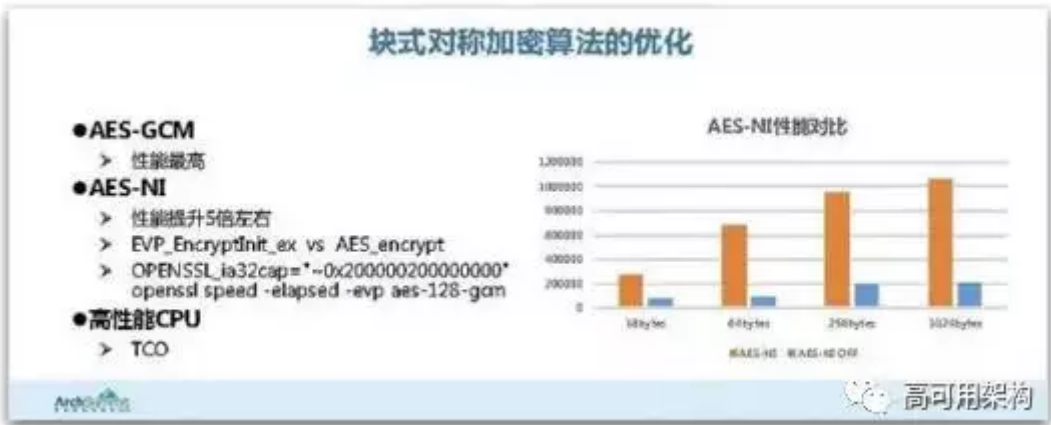
验证内容包括 3.2 Signed Response Acceptance Requirements[4]：

- 验证证书一致
在OCSP Response 中所指的证书和请求中所指证书一致。
- 验证签名有效
OCSP Response 中的签名有效。
- 签名者身份合法
 - 签名者的身份和相映应接受请求者匹配。
 - 签名者正被授权签名回复。
- 验证时间
 - 表示状态被认为是正确的时间（此次更新）足够新。
 - 如果有的话，下次更新的时间应该晚于现时时间。

对比未开启 OCSP 的访问，开启 OCSP 验证后，完整的握手建连过程中，请求时长会增加 5ms 左右的时间，用来传输 Certificate Status 数据及验证 OCSP response 是合法可信的。

加密算法优化调研

出于减少重复劳动的原因，这里直接给出一份 ArcSummit 上腾讯大牛分享过的一份优化测试文档里的数据：[腾讯HTTPS性能优化实践](#)



对于老旧机型， `chacha20` 性能有显著的提升。



参考资料

这篇文章的形成参考了很多大牛先前的实践经验与总结。感谢这些 **巨人的肩膀** 让我们可以有更广阔的视野：

- im.ququ 的小站关于 http https 的文章与心得 [5]
- HTTP/2 探索第一篇：概念 [6]
- [腾讯 HTTPS 性能优化实践](#)
- 开始使用 ECC 证书 [7]
- Keyless SSL: The Nitty Gritty Technical Details [8]
- ECDH 算法概述（CNG 示例） [9]
- Man-in-the-middle attack [10]

我是谁：李子昂，WX: Regin (ID:Regin_110)

干啥的：美图公司架构通讯核心研发工程师

还有啥：美图疯狂招人中，C，Golang，Java，PHP.....

职责：主要负责**美图后端业务研发**、**编解码研发**、**直播相关研发**、**虚拟化**等各种方向，优秀的人在这里总会找到属于你的位置，

工作地点：北京、厦门、深圳随意挑选。快到碗里来！

简历投递：yt@meitu.com 邮件请署名投递方向。

- [1] http://blog.csdn.net/arthur_killer/article/details/71405249
- [2] <http://www.ietf.org/mail-archive/web/tls/current/msg05593.html>
- [3] <https://github.com/arthurkiller/wrk-go>
- [4] <https://www.ietf.org/rfc/rfc2560.txt>
- [5] <https://imququ.com/post/series.html>
- [6] <https://www.qcloud.com/community/article/164816001481011799>
- [7] <https://imququ.com/post/ecc-certificate.html>
- [8] <https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/>
- [9] [https://msdn.microsoft.com/zh-cn/library/cc488016\(v=vs.90\).aspx](https://msdn.microsoft.com/zh-cn/library/cc488016(v=vs.90).aspx)
- [10] https://en.wikipedia.org/wiki/Man-in-the-middle_attack

推荐阅读

- [WebP已经适合主流使用？美图图像选型评测及优化历程](#)
- [我们如何使用HAProxy实现单机200万SSL连接](#)
- [HTTPS环境使用第三方CDN的证书难题与最佳实践](#)