**摘要**

QUIC加密协议是QUIC的一部分，它为连接提供了传输安全性。QUIC加密协议是 **注定要消亡**的。未来它将由TLS 1.3替代，但在TLS 1.3 最终启用之前QUIC需要一个加密协议。  
  
借助于当前的QUIC加密协议，当客户端已经缓存了关于服务器的信息时，它可以无需往返就建立一个加密的连接。TLS，相反地，至少需要两次往返（算上TCP的3次握手）。QUIC握手应该比普通的TLS 握手（2048-bit RSA）高效大约5倍，而且安全等级更高。

**源地址欺骗**

Internet上的协议很少能在无需至少一个初始化往返的情况下就可以工作的。大多数协议由于TCP而需要一个往返，基于TLS的协议在应用数据可以传输前，需要至少额外的一个或更多往返。

这些往返都交换nonces：在TCP的情况下是序列号（或SYN cookies），在TLS的情况下是密码学随机值（client\_random 和 server\_random）。TCP nonce防止IP地址欺骗，而TLS nonce防止重放攻击。任何想要寻求减少往返的协议都不得不以某种方式解决这两个问题。

作为一个反例，DNS是一个没有任何初始化往返的协议，因此它不得不自己处理IP地址欺骗和重放攻击。DNS简单地忽略IP地址欺骗，并因此使镜像DDoS攻击成为一个真正的问题。对于重放的保护，DNSSEC依赖时钟同步和短时的签名。根据设计，这允许有限时间内的重放，由于那与DNS的缓存语义相互协调。但那不是重放保护的严格形式，因为重放是被允许的。

在QUIC中，我们分开处理这两个问题。

IP地址欺骗问题通过给客户端分配一个，即期的，“源地址token”来处理。从客户端的视角来看，这是透明的字节串。从服务器的视角来看，它是一个认证的加密块（比如 AES-GCM），其中包含，至少，客户端的IP地址，和一个服务器的时间戳。服务器将只为给定的IP给那个IP发送一个源地址token。客户端收到token被视为对IP地址拥有所有权的证明，与TCP序列号的接收 所采用的方式一样。

客户端可以在未来的请求中包含源地址token以证明对它们的源IP地址的所有权。如果客户端的IP地址变了，则token也过时了，或者客户端没有token，则服务器可以拒绝连接，并返回一个新的token给客户端。但是如果客户端的IP地址保持不变，则它可以复用源地址token以避免获取一个新token所需的网络往返。

Token的生命周期主要与服务器有关，但由于源地址token是不记名token，它们可能被盗取并被复用以绕过基于IP地址的限制。（尽管攻击者将无法收到响应。）源地址token还可以被收集，并可能在IP地址的所有权发生变化之后使用（比如，在DHCP池中）。简少token的寿命，以减少在无需额外往返的情况下处理的请求的数量为代价来改善这两个问题。

源地址token，不像TCP序列号的交换，不要求源表现出连续的接收发送到源IP地址的分组的能力。这允许源地址token被用于持续地请求来自于服务器的业务，即使下行链路已经饱和，其丢包率足够高以至于不能建立TCP连接。这种 “自 DOS” 攻击可用于 DOS 相同下行链路中的其他用户。

然后，我们注意到 [类似的技巧实际上可能用于TCP](http://www.cs.umd.edu/~capveg/optack/optack-extended.pdf)，因此QUIC并没有在这方面明显地使事情变得更糟。确实，一旦连接建立好，QUIC在包中包含了一个熵位，并要求接收者发送它们声称已经收到的熵的散列值 - 从而解决TCP的问题。

为了最小化延迟，服务器可以动态地决定放松源地址限制。可以想象服务器跟踪来自不同IP地址的请求的数量，并且只有当“未请求”连接的数量超过全局或某个IP范围的限制时，才需要源地址token。这可能是有效的，但不清楚这是否是全球稳定的。如果大量的QUIC服务器实现了这种策略，则大量的镜像DDoS攻击可以在它们之间分割，使得任何一台服务器都不会达到攻击阈值。

**重放攻击**

在TLS中，每一方都生成一个随机数，通过强制它们在密钥中包含（假定在所有时间唯一）该值，用于确保另一方是新的。没有往返，客户端仍然可以包括一个随机值以确保服务器是新的，但服务器没有机会为客户端这样做。

在没有来自服务器的输入的情况下提供重放保护基本上是非常昂贵的。它需要服务器端的一致状态。尽管如果服务器是单台机器的话这是合理的，但现代的网站都遍布全世界。

因此，QUIC在服务器的第一次应答之前不为客户端的数据提供重放保护。这依赖应用去确保这样的信息在被攻击者重放时是安全的。比如，在Chrome中，只有GET请求在握手确认前发送。

**握手开销**

在TLS中，服务器基于客户端广告的它们支持的参数为每个连接选择连接参数。在QUIC中，服务器的首选项完全是枚举的和静态的。它们与Diffie-Hellman公共值一起捆绑到一个“服务器配置”中。这个服务器配置含有一个过期时间，并由服务器的私钥签名。由于服务器配置是静态的，因而不是每个连接都需要签名操作，而是单个签名足以满足许多连接。

使用Diffie-Hellman可用的连接密钥。服务器的 Diffie-Hellman 值在服务器配置中发现，客户端在它的第一个握手消息中提供。由于服务器配置必须保留一段时间，以允许0-RTT握手，这给连接的前向安全性设置了上限。既然服务器记录了服务器配置的 Diffie-Hellman 密钥，则如果它们泄漏的话用那个服务器配置加密的数据可能被解密。

这样QUIC提供了两个层面的保密：来自于客户端的初始数据使用服务器的服务器配置中的 Diffie-Hellman 值加密，这可能持续几天。一接收到连接，服务器就用一个 短暂Diffie-Hellman 值来响应，然后连接被重新计算密钥。

（相对于前向安全的TLS连接，这可能似乎提供更少的前向安全性。然而，为了避免往返，通常在大规模部署中都会启用TLS Session Tickets。SessionTicket 密钥足以解密连接，但为了恢复的有效性，它必须保持合理的时间段 - 通常是几天。SessionTicket密钥和服务器配置密钥类似，且有效的安全性实际上比QUIC要高，因为它的前向安全模式更优越。）

单个连接是通常的前向安全性的范围，但是用于一个单独的连接的短暂的密钥，和在 60 秒内用于所有连接的密钥的安全性的差异是微不足道的。因此我们可以在小的时间跨度中将服务器的 Diffie-Hellman 密钥生成分摊在所有的连接上。

（由于服务器配置和 Diffie-Hellman 私有值是服务器为了处理QUIC连接所需的所有东西，证书的私钥从不需要放在服务器上。相反，短期证书的一种形式可以通过签署短期服务器配置并仅安装在服务器上来实现。）

如果我们设 **S** 是一个密钥操作（比如RSA解密），**P** 是一个公钥操作（比如 RSA加密），**F** 是一个 Diffie-Hellman，定点的，标量乘法，而且 **A** 是一个任意点，标量乘法则：

1. TLS，非前向安全握手：服务器，1**S** (1100µs)；客户端，1**P** (34µs)。
2. TLS，前向安全握手：服务器，1**S** + 1**F** + 1**A** (1301µs)；客户端，1**P** + 1**F** + 1**A** (235µs)。
3. QUIC：服务器，2**A** (100µs)；客户端，1**F** + 2**A** + 1**P** (184µs)。

（客户端验证证书链的操作没有包含在内。）

如果我们为这些中的每个选择公共基元（RSA 2048用于公共和私有操作，ECDH P-256用于TLS前向安全，Curve25519用于QUIC的），那么我们获得在i7-3770S上的括号中的示例时间。如果 QUIC使用 P-256，则服务器时间将是 300µs，客户端将是385µs，所以相当多的收益来自于更好的基元。

粗略统计TLS会话恢复率大约为 50%，但 QUIC 不包含显式地会话恢复。然而，它可以在不在协议中支持恢复的情况下获得许多恢复的益处，通过使客户端和服务器维护一个 Diffie-Hellman 结果的缓存。只要客户端没有旋转其临时密钥，这个可选的缓存就可以消除使用同一服务器多次握手的计算负担。如果我们假设TLS的恢复率为 50%，并假设QUIC缓存不做任何事，则相对于简介中提到的TLS，我们获得大约 5x 的速度提升。

**Wire协议**

QUIC是一个数据报协议，一旦密钥建立，则每个数据报的完整载荷（在UDP层之上）都是被认证和加密的。底层的数据包协议为加密层提供了可靠的发送方式，任意大小的消息。这些消息具有一个统一的，键值格式。

键是 32-bit **tags**。这试图在魔术数字注册表的粗暴和字符串的冗长之间提供一个平衡。就Wire协议而言，这些是不透明的，32位的值，在本文中，**tags** 通常将是 EXMP。虽然它是一个字符串，但是它只是值 0x504d5845 的助记符。该值，是小尾端的，是ASCII字符串 E X M P。

如果 **tag** 是ASCII，但它少于四个字符，则就好像剩余的字符是NUL。因此 EXP 对应于 0x505845。

如果 **tag** 值包含超出ASCII范围的字节，则它们将以十六进制格式写入，比如，504d5845。

除非另有说明，否则所有值都是小尾端的。

握手消息的组成为：

1. 消息的标签（tag）。
2. 包含标签值对数量的uint16。
3. 发送时的两个值为零的填充字节，接收时要被忽略。
4. 一系列 uint32 标签 和 uint32 结束偏移，每个标签值对一个。标签必须是严格地单调递增的，并且结束偏移必须是单调的非递减。结束偏移给出了偏移量，从值数据的开始处算起，到超出那个标签的数据的结尾处一个字节的位置结束。（因此最后一个标记的结束偏移包含值数据的长度）。
5. 值数据，无填充连接。

标签值格式允许在只有一小部分数据被验证后对标签进行有效的二分搜索。标签严格单调的要求也消除了关于重复标签的任何歧义。

尽管当前 32 位的长度超出了需要，16位长度存在不足以处理更大的后量子值的风险。

任何消息可以包含一个填充 (PAD) 标记。这些可以用来打败流量分析。此外，我们可以为客户端的 hellos 定义一个全局的最小大小以限制放大攻击。小于最小值的客户端hello将需要PAD标记来弥补差异。

**客户端握手**

客户端握手的流程如图1。概念上来说，QUIC中的所有握手都是 0-RTT的，只是它们中的一些会失败，需要重试。

为了执行0-RTT握手，客户端需要具有已被验证为可信的服务器配置。最初，我们假设客户端不知道任何关于服务器的东西，因此，在可以尝试握手之前，客户端将发送“inchoate” 客户端 hello 消息以从服务器引出服务器配置和真实性证明。在客户端收到其需要的所有信息前，可能会有几轮 inchoate 客户端 hellos，因为服务器可能不愿意向未经验证的IP地址发送大量的真实性证明。

Client hello 具有消息标记CHLO，并且以其初始形式包含以下标记/值对：

* **SNI** Server Name Indication (服务器名称指示)（可选的）：服务器的完全限定DNS名称，规范化为小写，没有尾随周期。国际化的域名需要被编码为 RFC 5890 中定义的 A-labels。SNI 标签的值不能是IP地址字面量。
* **STK** 源地址令牌 (Source-address token)（可选的）：服务器先前提供的源地址令牌（如果有）。
* **PDMD** 证明需求 (Proof demand)：描述客户端可接受的证明类型的标签列表，按照优先顺序。目前只定义了X509。
* **CCS** 公共证书集 (Common certificate sets)（可选的）：一系列 64 位，[**FNV-1a**](http://isthe.com/chongo/tech/comp/fnv/#FNV-param) 散列的客户端拥有的公共证书集 。（参考关于证书压缩的小节。）
* **VER** 版本：单个标签，反映客户端在第一个数据传输中的每个QUIC数据包中通告的协议版本。如果发生了版本协商，则这个字段被设为客户端使用的第一个版本。如果包中的版本不等于标签中的版本，则服务器需要验证服务器不支持标签中的版本以防御降级攻击。
* **XLCT** 客户端期望服务器使用的叶证书的64位，[**FNV-1a**](http://isthe.com/chongo/tech/comp/fnv/#FNV-param) 哈希值。证书的完整内容将被加进 HKDF。如果存在缓存的证书，则首个这样的条目应与此字段的值一致。

（QUIC的其它部分可以定义客户端和服务器的 hellos 中包含的额外标签。比如，流的最大个数，拥塞控制参数等等。然而，那些标签不在本规范中定义。）

作为对客户端 hello 的响应，服务器将发送一个拒绝消息，或一个服务器 hello。服务器 hello 表示一个成功的握手，并且永远不会从初始客户端 hello 产生，因为它不包含足够的信息来执行握手。拒绝消息包含客户端可用以在后面执行更好的握手的信息。

拒绝消息具有 **REJ** 标签，且包含如下的标 签/值 对：

* **SCFG** 服务器配置(Server config)（可选的）：包含了服务器的序列化的配置的消息。（在下面描述。）
* **STK** 源地址令牌 (Source-address token)（可选的）：客户端应该在未来的客户端 hello 消息中回显的透明字节串。
* **SNO** 服务器随机数 (Server nonce)（可选的）：服务器可以设置一个随机数，客户端应该在任何未来的（完整的）客户端 hello 消息中回显此随机数。这允许服务器在没有触发寄存器的情况下操作，而客户端在时钟偏斜的情况下连接。
* **STTL** 服务器配置有效的持续时间，以秒计，
* **ff545243** 证书链（可选的）：服务器的证书链。（参考关于证书压缩的小节。）
* **PROF** 真实性证明（可选）：在 X.509 的情况下，服务器配置的叶子证书的公钥签名。当前这个签名的格式由公钥的类型固定：

| **RSA** | **RSA-PSS-SHA256** |
| --- | --- |
| ECDSA | ECDSA-SHA256 |

签名通过如下方式计算：

1. 标签 “QUIC server config signature”
2. 下一个字段中哈希的32位长度的字节数（即8）
3. CHLO 的 SHA256 哈希
4. 一个 0x00 字节
5. 序列化的服务器配置

尽管拒绝消息的所有元素是可选的，但是服务器必须允许客户端进行。比如，如果客户端不存在源地址令牌，且服务器不希望发送服务器配置给一个未经验证的 IP 地址，则服务器必须以一个源地址令牌来响应以使客户端接下来的握手尝试更加成功。

一些标记以十六进制而不是以ASCII符号指定。这是因为标签被构造为使得它们将在消息的开始或结束到来。回想一下，标记，作为数字，是以小尾端序写入线上的。

包含熵的标签被移动到消息的开始，因为服务器可能不维护状态，因此可以处理重复的客户端 hello 两次。如果拒绝消息分组丢失，并且熵字段跨越分组边界，则客户端可能错误地组合它们。

大标记（到目前为止是证书链）被移动到消息的结尾，使得它们不会延迟可能足够的其它字段的接收。

服务器配置包含序列化的服务器首选项，并采用具有标签 SCFG 的握手消息的形式。它包含如下的标 签/值 对：

* **SCID** 服务器配置 ID：这个服务器配置的透明的，16字节标识符。
* **KEXS** 密钥交换算法：标记列表，以优先顺序，指定了服务器支持的密钥交换算法。定义了以下标记：

| **C255** | **Curve25519** |
| --- | --- |
| P256 | P-256 |

* **AEAD** 验证加密算法：标记列表，以优先顺序，指定了服务器支持的 AEAD 基元。定义了以下标记：

| **AESG** | **具有 12 字节标记和 IV 的AES-GCM。IV 的头四个字节取自于密钥推导，而后 8 个则是分组序列号。** |
| --- | --- |
| S20P | Salsa20 with Poly1305。(暂时还没有实施。) |

* **PUBS** 公共值的列表，24 位，小尾数长度前缀，与 KEXS 相同的顺序。P-256 公共值，如果有的话，被编码为X9.62格式的未压缩点。
* **ORBT** 轨道（Orbit）：一个8字节的不透明值，用于标识触发寄存器（残留）。
* **Expiry** 到期（Expiry）：以 UNIX epoch 秒计数的服务器配置 64 位到期时间。
* **VER** 版本：服务器支持的版本标记的列表。底层的 QUIC 包协议有版本协商。服务器支持的版本是签名的服务器配置的镜像，以确保没有降级攻击的发生。

一旦服务器接收了服务器配置，且已经认证了它并验证了证书链和签名，它可以通过发送完整的 client hello 来执行一个不是设计为失败的握手。完整 client hello 包含与初始 client hello 相同的标签，加上几个其他的：

* **SCID** 服务器配置 ID：客户端使用的服务器配置 ID。
* **AEAD** 验证加密：被使用的 AEAD 算法的标签。
* **KEXS** 密钥交换：被使用的密钥交换算法的标签。
* **NONC** 客户端随机数：由 4 字节的时间戳（大尾端，UNIX epoch 秒），8 字节的 服务器轨道，和 20 字节的随机数组成的 32 字节数。
* **SNO** 服务器随机数（可选的）：回显的服务器随机数，如果服务器提供了的话。
* **PUBS** 公共值：对于给定的密钥交换算法，客户端的公共值。
* **CETV** 客户端加密标签值（可选的）：序列化消息，以在 client hello 中指定的 AEAD 算法加密，并且具有以下面 CETV 部分中指定的方式导出的密钥。此消息将包含进一步的加密的标签值对，指定客户端证书，ChannelID 等。

发送了完整的 client hello 之后，客户端拥有用于连接的非前向安全密钥，因为它可以计算来自服务器配置的共享值和PUBS中的公共值。（有关密钥推导的详细信息，请参见下文。）这些密钥被称为初始密钥（而不是稍后的前向安全密钥）且客户端应该用这些密钥加密未来的包。它还应该配置数据包处理以接受使用这些密钥以锁定方式加密的数据包：一旦已经接收到加密分组，则不应接受另外的未加密分组。

在此时，客户端可以自由地开始向服务器发送应用程序数据。实际上，如果它希望实现0-RTT，则它必须在等待服务器的答复之前开始发送。

数据的重传发生在握手层下面的层，然而该层必须仍然知道加密的改变。新的分组必须使用初始密钥来传输，但是如果 client hello 需要重传，则必须以明文的方式重传。分组发送层必须知道哪个安全级别最初用于发送任何给定分组，并且小心不要使用更高的安全级别，除非对端已经确认拥有这些密钥（即通过使用该安全级别发送分组）。

服务器将接受或拒绝握手。服务器拒绝 client hello 的情况下，它将发送 REJ 消息，并且使用初始密钥发送的所有分组必须被认为丢失并且需要在新的初始密钥下重传。因此，在 server hello 或 拒绝 待定时，客户端应该限制未完成的数据量。

在理想的情况下，握手成功，服务器返回一个 server hello 消息。此消息具有标记 SHLO，使用初始密钥加密，除了为拒绝消息定义的标记/值对之外，还包含以下标记/值对：

* **PUBS** 客户端用于密钥交换算法的临时公共值。通过手中的临时公共值，双方都可以计算前向安全密钥。（见关于密钥推导的部分。）服务器可以立即切换为使用前向安全密钥发送加密数据包。客户端必须等待收 server hello。（注意：我们正在考虑让服务器等待，直到它在发送任何自己之前接收到前向安全数据包。如果 server hello 数据包被丢弃，这避免了停顿。）

**密钥推导**

密钥材料由经过散列函数 SHA-256 的基于 HMAC 的密钥导出函数（HKDF）生成。HKDF（在 [RFC 5869](https://tools.ietf.org/html/rfc5869) 中描述）使用 [NIST SP 800-56C](http://csrc.nist.gov/publications/nistpubs/800-56C/SP-800-56C.pdf) 中描述的已批准的两步密钥导出过程。

**第 1 步：HKDF 提取**  
密钥协议的输出（在Curve25519和P-256的情况下为32字节）是预主密钥，后者是HKDF-提取 函数的输入密钥材料（IKM）。盐 输入是客户端随机数，后跟服务器随机数（如果有的话）。HKDF-提取 输出伪随机密钥（PRK），其是主密钥。 如果使用SHA-256，主密钥为32字节长。

**第 2 步：HKDF 扩展**  
**PRK** 输入是主密钥。**info** 输入（上下文和应用程序特有信息）是以下数据的级联：

1. 标签 “QUIC key expansion”
2. 一个 0x00 字节
3. 来自数据包层的连接的GUID。
4. client hello 消息
5. 服务器配置消息
6. DER 编码的叶子证书的内容

密钥材料按以下顺序分配：

1. 客户端写密钥。
2. 服务器写密钥。
3. 客户端写 IV。
4. 服务器写 IV。

如果任何原语需要少于密钥材料的整个字节数，则丢弃最后一个字节的剩余部分。

当推导前向安全密钥时，使用相同的输入，除了info使用标签“QUIC前向安全密钥扩展 (QUIC forward secure key expansion)”。

当推导服务器的初始密钥时，它们必须是多样化的，以确保服务器能够向HKDF提供熵。

**第 1 步：HKDF 提取**

服务器写密钥加上来自找到的轮的服务器写IV的级联是用于 HKDF-Extract 函数的输入密钥材料（IKM）。盐输入是多样化随机数。HKDF-Extract 输出一个伪随机密钥（PRK），它是多样化密钥。如果使用 SHA-256 的话，多样化密钥是 32 字节长的。

**第 2 步：HKDF 扩展**  
PRK 输入是多样化密钥。info 输入（上下文和应用特有信息）是标签”QUIC 密钥多样化 (QUIC key diversification)”。

密钥材料按以下顺序分配：

1. 服务器写密钥。
2. 服务器写 IV。

**客户端加密的标签值**

client hello 可能包含一个 CETV 标签，以描述客户端证书，ChannelIDs 和 client hello 中的其它非公有数据。（那与 TLS 相反，它以明文发送客户端证书。）

CETV 消息以 client hello 中的 AEAD 序列化和加密。密钥是以与连接的密钥相同的方式推导的（参考上面的 密钥推导），除了 info 使用标签“QUIC CETV 块（QUIC CETV block）”。推导中所用的 client hello 消息是无 CETV 标记的 client hello。当随后推导连接密钥时，所使用的 client hello 将包含 CETV 标记。

AEAD 随机数总是 0，它是安全的，这是因为只有一个消息曾以该密钥加密。

通过对 CETV 密钥推导中所使用的 HKDF **info** 输入签名，来完成对客户端证书和 ChannelID 两者所需的私钥的拥有。

CETV 消息可以包含如下的标签：

CIDK ChannelID 密钥（ChannelID key）（可选的）：一个 32字节对，大尾端数字，一起描述一个 (x, y) 对。这是 P-256 曲线上的一个点和一个 ECDSA 公钥。

CIDS ChannelID 签名（ChannelID signature）（可选的）：一个 32字节对，大尾端数字，一起描述一个 HKDF 输入的 ECDSA 签名的 (r, s) 对。

**证书压缩**

在 TLS 中，证书链是无压缩传输的，并占用了完整握手中的绝大多数字节。在 QUIC 中，我们希望能够通过压缩证书来避免一些往返。

证书链是一系列的证书，就这一节的目的而言，是透明的字节串。叶子证书总是链中的第一个，且从不应该包含根 CA 证书。

当在一个拒绝消息的 CRT\xFF 标签中序列化一个证书链的时候，服务器认为那些信息客户端已经有了。这些先验知识可以来自于两种方式：拥有一般中间证书，或者缓存了之前与相同服务器交互时的证书。

前者表示为 client hello 的 CCS 标签中的一系列 64-bit [FNV-1a](http://isthe.com/chongo/tech/comp/fnv/#FNV-param) 哈希值。如果客户端和服务器共享了至少一个一般证书集合，则可以简单地引用它们中存在的证书。

缓存的证书表示 client hello 的 CCRT 标签中的 64-bit [FNV-1a](http://isthe.com/chongo/tech/comp/fnv/#FNV-param) 哈希值。如果任何一张依然在证书链中，则它们可由哈希值代替。

任何剩余的证书以一个预共享的由前两个方法指定的证书和取自于 Alexa top 5000 的证书中的字符串组成的字典经 gzip 压缩。

具体的表示方式被放置在拒绝消息的CERT标签中，并具有以下 TLS 表示风格中的 Cert 结构格式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | enum { end\_of\_list(0), compressed(1), cached(2), common(3) } EntryType;  struct {  EntryType type;  select (type) {  case compressed:  // nothing  case cached:  opaque hash[8];  case common:  opaque set\_hash[8];  uint32 index;  }  } Entry;  struct {  Entry entries[];  uint32 uncompressed\_length;  opaque gzip\_data[];  } Certs; |

（回忆一下，QUIC 中的数字是小尾端的。）

entries 列表以一个类型为 end\_of\_list 的 Entry 结束，而不是  
TLS 中常见的长度前缀。gzip\_data 扩展到值的末尾。

gzip，预共享字典包含类型为 compressed 或 cached 的证书，以相反的顺序连接，其后是此处未提供的 ～1500 个字节的公共子字符串。

**未来方向**

1. ChannelID 非常有可能从协议层中移除，相反，加密握手将产生可以在较高层签名的通道绑定值。
2. Trevor Perrin 已经指出，服务器可以返回一个加密票证，其中包含客户端可以在未来的连接中回显给服务器的 Hash (前向安全保密)。对于那些握手，这将节省一次 Diffie-Hellman 操作。
3. 服务器应该能够向客户端指示，它们应该在发送应用数据之前等待直到前向安全密钥建立。
4. 为了避免 server hello 包造成的队首阻塞，服务器可以避免发送前向安全数据，直到客户端确认接收到 server hello。（比如：通过自身发送前向安全包。）

**感谢**

多谢 Trevor Perrin， Ben Laurie 和 Emilia Käsper 的有价值的反馈。