使用Proverif分析TLS 1.3协议

文章内容简介

TLS(传输层安全)协议是一种广泛使用的网络协议，它主要用来建立服务器和客户端之间的安全通道。本文对TLS 1.3的18草案就行详细解析，并用proverif对其进行建模已经进行安全性论证。本文已上传到知乎：https://zhuanlan.zhihu.com/p/534666160，CSDN：https://blog.csdn.net/feidun01/article/details/125494977?spm=1001.2014.3001.5501

密码学基础知识

Diffie-Hellman密钥交换协议

在加密通信过程中，有时要用到共享密钥(shared key)，这个密钥为通信双方所共享，但是要保证不可以被攻击者发现。Diffie-Hellman密钥交换协议确保即使攻击者截获通信信道上的所有信息，都无法在可以接受的时间内算出它们的共享密钥。

该协议用到了原根的一个性质：假设q是p的一个原根，则q的幂构成模p的简化剩余系，即q的幂模p可以取到1到p-1的任意一个值。具体证明可以参见https://www.cnblogs.com/philolif/p/number-theory-6.html。由求q的幂模p的值很容易，但是已知这个值，反过来求指数的过程计算周期却很长，基本只能用试错法，计算时间复杂度和p-1同量级。如果p非常大，那么逆向求解是无法有效做到的。

通信双方首先约定一个大素数p和它的原根g，这两个量是无需保护的。接下来A生产随机数a，将$g^a(modp)$通过公共信道发给B，接下来B生成随机数b，将$g^{ab}(modp)$作为密钥，并将g^b发给A，随后A将$(g^b)^a = g^{ba}(modp)$作为密钥。由于$g^{ab} = g^{ba}$，A和B便得到公共密钥。

攻击者从信道上可以截获g^a和g^b，然而从这两者还原g^{ab}是无法做到的，这就确保了协议的安全性。

然而，到此为止我们仅赋予了攻击者监听信息的权力。如果再考虑攻击者综合利用信息、伪造并重发送的情况，之前提到的通信方式仍然是不安全的：攻击者可以使用**中间人攻击来**获得共享密钥。为应对中间人攻击，可以使用数字签名。

接下来使用Proverif工具来对Diffie-Hellman密钥交换协议进行建模。这里不考虑数字签名，因为在最终的验证程序中数字签名额外生成并与通信消息绑在一起。定义dh_ideal(element,bitstring)为求幂函数，将element类型的参数求对另一个参数的幂，得到另一个element类型的值。这里的element可以理解成完全剩余系中的元素。除此之外，再考虑一种特殊情况：g^a模p只有极少数的可能值，以至于可以通过试错法破解共享密钥。出现这种情况的原因可能是g不是原根。不妨考虑极端情况：g^a模p只有一个可能值，记为BadElement。为了区分两种加密方式，我们再定义一种方法dh_exp，它接受一个参数标记使用正常的DH算法(StrongDH)或者使用上面提到的出现BadElement的方法(WeakDH)。如果是StrongDH，在底数是BadElement时返回BadElememt，否则按dh_ideal进行计算；如果是WeakDH，则返回BadElememt。此外为保证$g^{ab} = g^{ba}$这一代数性质，需要定义一个方程确保dh_ideal(dh_ideal(G,x),y) = dh_ideal(dh_ideal(G,y),x)。

完整代码如下：

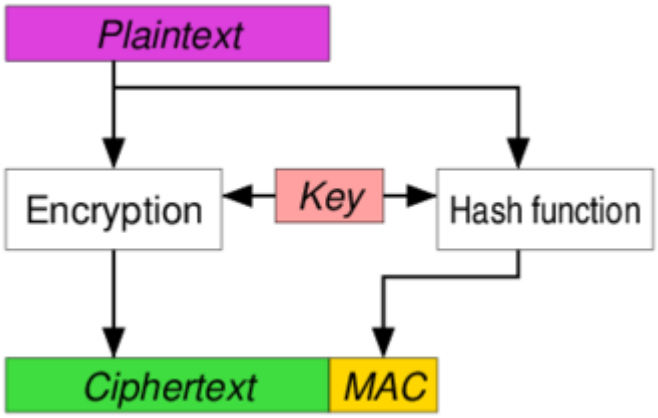
```
(*****************************************************)
(* Diffie-Hellman with small/bad subgroup attacks. See Logjam, Cross-Protocol *)
(*****************************************************)

type group.
```

```
const StrongDH: group [data].
const WeakDH: group [data].

type element.
fun e2b(element): bitstring [data].
const BadElement: element [data].
const G: element [data].

fun dh_ideal(element,bitstring):element.
equation forall x:bitstring, y:bitstring;
     dh_ideal(dh_ideal(G,x),y) =
     dh_ideal(dh_ideal(G,y),x).

fun dh_exp(group,element,bitstring):element
reduc forall g:group, e:element, x:bitstring;
      dh_exp(WeakDH,e,x) = BadElement
otherwise forall g:group, e:element, x:bitstring;
      dh_exp(StrongDH,BadElement,x) = BadElement
otherwise forall g:group, e:element, x:bitstring;
      dh_exp(StrongDH,e,x) = dh_ideal(e,x).

letfun dh_keygen(g:group) =
       new x:bitstring;
       let gx = dh_exp(g,G,x) in
       (x,gx).
```

AEAD加密

相比传统加密算法，AEAD加密算法支持解密前进行验证，以得知密钥正确性。为了便于理解，此处选一种AEAD加密分析进行说明。

E&M (Encryption and MAC)：首先对密文进行加密，然后使用同一密钥进行MAC运算，将Concat(Ciphertext,MAC)发给对方。对方收到消息后先解密，然后将解密的信息重新做MAC运算，如果和收到的MAC值相同，则验证成功，说明密钥正确。(下图选自wiki)



有时，可以添加nonce（如序列号）唯一地标定每次加密过程。

在安全认证中，我们需要再定义一种弱AEAD加密算法，它允许对加密的数据进行修改，定义相应方法为aead_forged，接受两个参数，第一个为预期修改的信息p，第二个是加密后的信息。输出的信息经过解密后变成p。除此之外，攻击者可以使用aead_leak方法直接由加密过的数据得到被加密的值。

Proverif建模如下：

```
(*********************************************************)
```

```
(* Authenticated Encryption with Additional Data *)
(* extended with with weak/strong algorithms: See Lucky13, Beast, RC4 *)
(******************************************************)

type ae_alg.
const WeakAE, StrongAE: ae_alg.
type ae_key.
fun b2ae(bitstring):ae_key [data].

fun aead_enc(ae_alg, ae_key, bitstring, bitstring, bitstring): bitstring. (*编码*)
fun aead_forged(bitstring,bitstring): bitstring.

fun aead_dec(ae_alg, ae_key, bitstring, bitstring, bitstring): bitstring (*解码*)
reduc forall a:ae_alg, k:ae_key, n:bitstring, p:bitstring, ad:bitstring;
    aead_dec(a, k, n, ad, aead_enc(a, k, n, ad, p)) = p
otherwise forall a:ae_alg, k:ae_key, n:bitstring, p:bitstring,
ad:bitstring,p':bitstring,ad':bitstring;
    aead_dec(WeakAE, k, n, ad, aead_forged(p,aead_enc(WeakAE, k, n, ad', p'))) = p.

fun aead_leak(bitstring):bitstring
reduc forall k:ae_key, n:bitstring, ad:bitstring, x:bitstring;
      aead_leak(aead_enc(WeakAE,k,n,ad,x)) = x.
```

Hash

可以将一个对象映射到某一个范围中的值。如果有两个不同对象映射到同一个值，则称为哈希冲突。一般来讲，可以通过检测两个值的Hash值是否相等，在不知道这两个具体值的情况下判断它们是否相等。和上面类似，我们可以定义一种“脆弱”的哈希算法，它很容易产生冲突。这样两个不相同的数可能被错误地被认证成相同。假设该算法下所有数据的哈希值全部被映射为collision。

哈希的proverif建模：

```
(******************************************************)
(* Hash Functions, including those with collisions. See SLOTH *)
(******************************************************)

type hash_alg.
const StrongHash: hash_alg [data].
const WeakHash: hash_alg [data].

const collision:bitstring [data].
fun hash_ideal(bitstring):bitstring.

fun hash(hash_alg,bitstring): bitstring
reduc forall x:bitstring;
      hash(WeakHash,x) = collision
otherwise forall x:bitstring;
      hash(StrongHash,x) = hash_ideal(x).
```

HMAC

Hash方法可以较好地用于验证，但生成的哈希值很容易伪造。HMAC可以理解为使用密钥进行哈希，具体定义为：

$$HMAC(K,m) = H((K' \; XOR \; opad)||H((K' \; XOR \; ipad)||m))$$

当K'大于某个长度时，K'=H(K)，否则K'=K。双竖线表示连接。opad和ipad解释如下：

- opad is the block-sized outer padding, consisting of repeated bytes valued 0x5c
- ipad is the block-sized inner padding, consisting of repeated bytes valued 0x36

当然在Proverif中，可以隐藏很多实现的细节。定义一个名为mac_key的类型用于标记HMAC中的密钥hmac_ideal输入密钥和待编码数据，输出编码后的数据。hmac在mac_key的基础上添加哈希算法作为第一个参数。如果是WeakHash，输出的永远都是collision，否则按hmac_ideal就行正常编码。

```
(*********************************************************)
(* HMAC *)
(*********************************************************)

type mac_key.
fun b2mk(bitstring):mac_key [data,typeConverter].

fun hmac_ideal(mac_key,bitstring): bitstring.

fun hmac(hash_alg,mac_key,bitstring):bitstring
reduc forall k:mac_key, x:bitstring;
      hmac(WeakHash,k, x) = collision
otherwise forall x:bitstring, k:mac_key;
      hmac(StrongHash,k, x) = hmac_ideal(k,x).
```

签名验证

签名方拥有一个私钥，将使用私钥对待签名数据进行加密，随后将加密后的签名和原数据发给检验方。检验方收到数据后，使用公钥对签名进行解密，如果解密后的数据和原数据相同，则说明数据为签名方所发。由于签名需要用到私钥，任何未得知私钥的攻击者都无法构造签名。

在Proverif中可以定义两个类型：公钥pubkey和私钥privkey。对于任意一个私钥，可以假设总可以找到对应的公钥，即定义pk(privkey)，返回pubkey。但反过来，基于安全性，不存在对应的sk(pubkey)，也就是说不能根据公钥就获取私钥。verify对使用私钥k对信息x的签名sign(k,x)进行验证。给定k对应的公钥pk(k)和x，如果签名信息是sign(k,x)，则验证成功，应该返回true。整体代码如下：

```
(*********************************************************)
(* Public Key Signatures *)
(*********************************************************)

type privkey.
type pubkey.
fun pk(privkey): pubkey.
const NoPubKey:pubkey.

(* RSA Signatures, typically the argument is a hash over some data *)

fun sign(privkey,bitstring):bitstring.
```

```
fun verify(pubkey,bitstring,bitstring): bool
reduc forall k:privkey, x:bitstring;
      verify(pk(k),x,sign(k,x)) = true.
```

HKDF密钥派生算法

HKDF是基于HMAC的密钥派生算法，它可以将初始密钥扩展成更强大的密钥。

主要分为两部：提取和扩展。提取是把密钥转化为固定长度的伪随机数，扩展是将提取到的伪随机数扩展到指定长度。

提取的具体过程可以用如下公式表示：
$HKDF-Extract(salt,IKM)=HMAC(salt,IKM)$得到PRK，其中salt是盐，IKM是原始密钥。

扩展可以用如下公式表示：
$HKDF-Expand(PRK,info,L)=OKM$，OKM是长度为L的密钥输出，info是可选上下文和应用程序特定信息。
首先计算一系列哈希值，记为T(n)，其中n从0到255。
T(0)=长度为0的空字符串
$T(1)=HMAC(PRK,T(0)|info|0x01)$
$T(2)=HMAC(PRK,T(1)|info|0x02)$
$T(3)=HMAC(PRK,T(2)|info|0x03)$
直到计算到T(255)。
我们将这些T(n)连接起来，记为$T=T(1)|T(2)|\cdots|T(N)$，OKM即是T的前L个字节。

在TLS 1.3中，还需要用到HKDF-Expand-Label和Derive-Secret，它们的定义如下：

```
HKDF-Expand-Label(Secret, Label, HashValue, Length) =
HKDF-Expand(Secret, HkdfLabel, Length)

struct {
      uint16 length = Length;
      opaque label<9..255> = "TLS 1.3, " + Label;
      opaque hash_value<0..255> = HashValue;
} HkdfLabel;
Derive-Secret(Secret, Label, Messages) =
HKDF-Expand-Label(Secret, Label,
Hash(Messages), Hash.Length)
```

HKDF的proverif实现：

```
(******************************************************)
(*  TLS 1.3 Key Schedule  *)
(******************************************************)

type label.
const client_finished, server_finished, master_secret,
      client_key_expansion, server_key_expansion: label.
const tls13_client_handshake_traffic_secret,
      tls13_server_handshake_traffic_secret,
      tls13_client_early_traffic_secret,
      tls13_client_application_traffic_secret,
```

```
      tls13_server_application_traffic_secret,
      tls13_key, tls13_iv,
      tls13_early_exporter_master_secret,
      tls13_exporter_master_secret,
      tls13_resumption_master_secret,
      tls13_resumption_psk_binder_key,
      tls13_finished: label.

fun tls12_prf(bitstring,label,bitstring): bitstring.

letfun prf(k:bitstring,x:bitstring) =
       hmac(StrongHash,b2mk(k),x).

letfun hkdf_extract(s:bitstring,k:bitstring) =
       prf(s,k).

letfun hkdf_expand_label(k:bitstring,l:label,h:bitstring) =
       prf(k,(l,h)).

letfun derive_secret(k:bitstring,l:label,m:bitstring) =
       hkdf_expand_label(k,l,hash(StrongHash,m)).

letfun kdf_0() = hkdf_extract(zero,zero).

letfun kdf_es(psk:preSharedKey) =
       let es = hkdf_extract(zero,psk2b(psk)) in
       let kb = derive_secret(es,tls13_resumption_psk_binder_key,zero) in
       (es,b2mk(kb)).


letfun kdf_k0(es:bitstring,log:bitstring) =
       let atsc0 = derive_secret(es, tls13_client_early_traffic_secret, log) in
       let kc0   = hkdf_expand_label(atsc0,tls13_key,zero) in
       let ems0   = derive_secret(es,tls13_early_exporter_master_secret,log) in
       (b2ae(kc0),ems0).


letfun kdf_hs(es:bitstring,e:bitstring) =
       hkdf_extract(es,e).

letfun kdf_ms(hs:bitstring,log:bitstring) =
       let ms =   hkdf_extract(hs , zero) in
       let htsc = derive_secret(hs, tls13_client_handshake_traffic_secret, log) in
       let htss = derive_secret(hs, tls13_server_handshake_traffic_secret, log) in
       let kch =  hkdf_expand_label(htsc,tls13_key,zero) in
       let kcm =  hkdf_expand_label(htsc,tls13_finished,zero) in
       let ksh =  hkdf_expand_label(htss,tls13_key,zero) in
       let ksm =  hkdf_expand_label(htss,tls13_finished,zero) in
       (ms,b2ae(kch),b2ae(ksh),b2mk(kcm),b2mk(ksm)).

letfun kdf_k(ms:bitstring,log:bitstring) =
       let atsc = derive_secret(ms, tls13_client_application_traffic_secret, log)
in
       let atss = derive_secret(ms, tls13_server_application_traffic_secret, log)
in
```

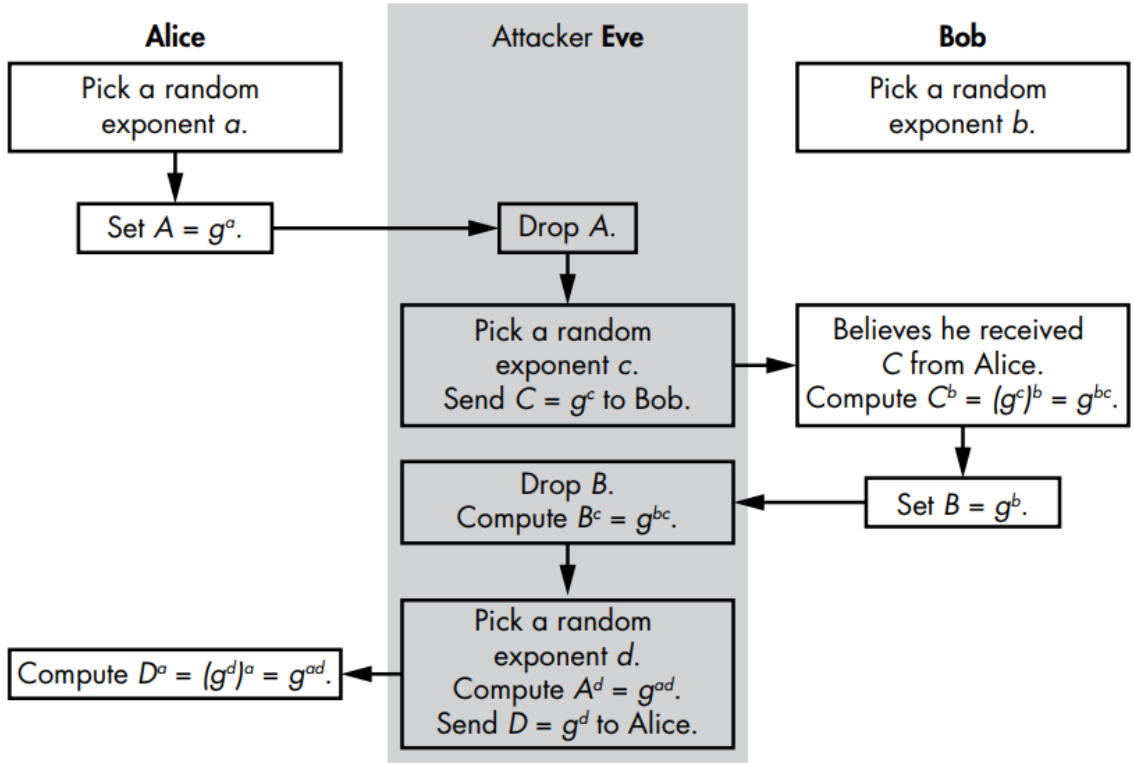
```
        let ems  = derive_secret(ms, tls13_exporter_master_secret, log) in
        let kc   = hkdf_expand_label(atsc,tls13_key,zero) in
        let ks   = hkdf_expand_label(atss,tls13_key,zero) in
        (b2ae(kc),b2ae(ks),ems).

letfun kdf_psk(ms:bitstring, log:bitstring) =
        derive_secret(ms,tls13_resumption_master_secret,log).
```

包含一些未提到的方法，稍后将会讲到，这里仅做参考。

中间人攻击

在通信协议中，可能出现第三者作为“中介”，伪造成对方和双方通信。在Diffie-Hellman密钥交换协议中，假设有一个攻击者Eve，他首先谎称自己是Bob和Alice建立通信，在得到Alice传过来的$A = g^a$后，扮成Alice和Bob通信，将$C = g^c$发给Bob，此时Bob生成共享密钥$C^B = g^{bc}$，并将$B = g^b$发给Eve，Eve收到B后，计算$C^c = g^{bc}$，作为其和Bob的共享密钥，同时选取随机数d，将$D = g^d$发给Alice，并将$A^d = g^{ad}$作为和Alice的共享密钥。这样Eve和Alice和Bob各有一对共享密钥。如果Alice通过共享密钥和Bob通信，则加密信息会先发给中间人Eve，Eve进行解密后使用和Bob的共享密钥再加密，将加密信息转发给Bob。这样Alice和Bob都认为消息是通过共享密钥直接发给对方的，而事实上它们的共享密钥并不相同，有一个中介Eve收取了它们所有的密钥并使消息都中中转给了Eve，Eve解密获得全部消息。

具体流程可参考下图：



TLS 1.3协议介绍

该部分参考TLS 1.3文档。

TLS的主要目的是为两台通信计算机提供一个安全的交流通道，这个通道应该保持以下三条属性：

- 服务器是经过认证的，客户端不能连到恶意服务器上。对于服务器来说，客户端只需要选择性认证。验证的方法可以是通过非对称密钥或者预共享对称密钥(pre-shared symmetric key)。
- 攻击者无法获得信道上传输的有效数据（无法解密关键数据）。

- 数据无法被攻击者修改。

TLS协议包含一下两个主要部分：

- 握手协议：对双方进行认证、协商加密方式和参数、建立共享密钥等等。
- 记录协议：使用握手协议中建立的参数进行通信。一次完整的通信可能被分成多次会话，每次都有独立的密钥，这样破解一次会话得到的信息并不能破解其他会话。

协议总览

```
      Client                                Server
Key   ^ ClientHello
Exch  | + key_share*
      | + pre_shared_key_modes*
      v + pre_shared_key*     ------>     ServerHello ^ Key
                                         + key_share* | Exch
                                    + pre_shared_key* v
                                {EncryptedExtensions} ^ Server
                                {CertificateRequest*} v Params
                                       {Certificate*} ^
                                 {CertificateVerify*} | Auth
                                           {Finished} v
                          <-------- [Application Data*]
      ^ {Certificate*}
 Auth | {CertificateVerify*}
      v {Finished} -------->
        [Application Data] <-------> [Application Data]
+表示在之前提到的消息
*表示可选项
{}表示被握手得到的traffic secret保护
[]表示被对应于特定会话的traffic_secret_N保护
```

握手过程可以分为三步：

- 密钥交换：建立共享密钥、选择密码学参数。随后的所有信息都是被加密过的。
- 服务器参数：建立其他握手参数，比如客户端是否被认证、应用层协议支持等等。
- 认证

密钥交换

在密钥交换的过程中，客户端发送ClientHello消息，这个消息包含一个随机数（防止降级攻击，稍后会提到）、提供的协议版本、各种密钥。服务器处理ClientHello并决定合适的密码学参数。随后发出ServerHello作为响应，这个响应中包含协商的参数以及服务器决定使用的密钥。

服务器参数

随后，服务器发送如下两条信息：

- 加密后的扩展(EncryptedExtensions)：对决定密码学参数不需要的扩展及其他特殊性的请求进行响应。
- 认证请求(CertificateRequest)：用于认证客户端。当客户端无需认证时，跳过这一步。

认证

- Certificate：当服务器不需要通过证书认证时被服务器忽略，当服务器没有给客户端发送认证请求时被客户端忽略。
- CertificateVerify：对整个握手过程进行签名。
- Finished：对于整个握手过程的MAC。这条消息提供密钥确认，将整个终端绑定到密钥上，在psk模式下也可以认证握手过程。

经历以上三个过程后，握手完成，此时可以传输应用层的数据。注意在客户端发送Finished之前不允许传输应用数据，这也是安全性论证的一个点，即论证TLS 1.3协议可以防止传输应用数据给未Finish认证的对象。

HelloRetryRequest

当客户端在ClientHello中传输的key_share不被服务器支持，服务器将响应HelloRetryRequest，客户端需要使用一个正确的key_share扩展再次发出请求。如果没有协商到合适的密码学参数（比如客户端的supported_groups和服务器不重合），服务器必须中止握手过程并发出警告。

完整过程如下：

```
Client                                   Server
ClientHello
+ key_share             -------->
                        <-------- HelloRetryRequest
                                     + key_share
ClientHello
+ key_share             -------->
                                     ServerHello
                                     + key_share
                            {EncryptedExtensions}
                            {CertificateRequest*}
                                   {Certificate*}
                             {CertificateVerify*}
                                       {Finished}
                    <-------- [Application Data*]
{Certificate*}
{CertificateVerify*}
{Finished}              -------->
[Application Data] <------->  [Application Data]
```

psk(预共享密钥复用)

当握手结束时，服务器可以给客户端一个预共享密钥。在之后的握手过程中，客户端可以复用该密钥。由于服务器已被认证，在再次使用PSK时，服务器无需发送Certificate和CertificateVerify。当客户端提供PSK时，应该再提供一个key_share扩展给服务器，以允许服务器丢弃psk开始一个完整的握手（如果需要的话）。服务器会响应一个预共享密钥，同时可以响应key_share扩展来进行(EC)DHE密钥的建立，提供前向安全性(forward secrecy，即攻击者获取某一会话所有密钥数据后仍不能对之前的所有密钥解密)。

握手协议详解

Part 1 密钥交换信息

密码学参数协商

从ClientHello可以得到以下四组参数：

- 客户端支持的AEAD算法和HKDF哈希对。
- (EC)DHE算法支持的群和key_share扩展。
- 签名算法。
- 预共享密钥和预共享密钥交换模式(psk_key_exchange_modes)。

服务器在ServerHello中指明如下的参数：

- 如果使用psk，则发送pre_shared_key表示选中的密钥。
- 如果不适用psk，则需要使用(EC)DHE和基于证书的认证来得到密钥。
- 如果使用(EC)DHE，服务器需要提供key_share扩展。
- 使用证书认证时，服务器需要发送证书和证书验证消息。

当服务器不能协商参数时，需要中止握手过程并发送handshake_failure或insufficient_security警告。

ClientHello

ClientHello数据块的C语言表示（选自TLS 1.3-18官方文档）：

```
uint16 ProtocolVersion;
opaque Random[32];
uint8 CipherSuite[2]; /* Cryptographic suite selector */
struct {
      ProtocolVersion legacy_version = 0x0303; /* TLS v1.2 */
      Random random;
      opaque legacy_session_id<0..32>;
      CipherSuite cipher_suites<2..2^16-2>;
      opaque legacy_compression_methods<1..2^8-1>;
      Extension extensions<0..2^16-1>;
} ClientHello;
```

以下是逐项解释：

- legacy_version：在旧版本的TLS中，用于表示客户端支持最高的版本。在TLS 1.2中，它被设为0x0303。在TLS 1.3中，仍然保持这个值，但客户端还要在supported_versions扩展中指明版本偏好。
- random：32字节的随机数，稍后将讲它的用途。
- legacy_session_id：在TLS 1.3之前，和会话恢复(session resumption)有关。在1.3版本中这个功能被预共享密钥取代，该域的值置零。
- cipher_suites：是加密选项的列表，包括密钥长度、HKDF哈希等等。当客户端包含服务器不支持的加密选项时，服务器必须将其忽略。

cipher_suites具体结构：

Components	Contents
TLS	字符串"TLS"
AEAD	AEAD算法
HASH	HKDF中使用的HASH算法
VALUE	两字节的ID

Value的取值：

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

- legacy_compress_methods：在TLS 1.3中必须设为0，否则引发illegal_parameter警告。
- extensions：扩展，下节讲。

ServerHello

结构体的C语言表示：

```
struct {
      ProtocolVersion version;
      Random random;
      CipherSuite cipher_suite;
      Extension extensions<0..2^16-1>;
} ServerHello;
```

以下是逐项解释：

- version：服务器选择的协议版本。
- random：服务器生成的独立于ClientHello.random的随机数。
- cipher_suite：服务器选择的加密方式。
- extensions：扩展列表。只能包含建立密钥通信需要的扩展，目前仅包括key_share和pre_shared_key。

对random域的说明

由于TLS 1.2及以下版本有较多的安全漏洞，攻击者可能会把协议版本设为1.2及以下来进行**降级攻击(downgrade attack)**。random域的作用便是防止这一攻击。事实上，不同版本的协议中random必须有不同的模式。支持TLS 1.2或更低协议版本的服务器仅支持最后8个字节为44 4F 59 4E 47 52 44 01的random，支持TLS 1.1或更低协议版本的服务器仅支持最后8个字节为44 4F 57 4E 47 52 44 00的random。如果服务器不支持降级使用，必须检查的random的后8位不能是上面两种情况中的一种，否则中止握手并发出illegal_parameter警告。

原文档中有HelloRetryRequest的讲解，但这部分对我们对协议的安全验证不重要，在此略过。

扩展

扩展的数据结构可以如下表示：

```
struct {
      ExtensionType extension_type;
      opaque extension_data<0..2^16-1>;
} Extension;
enum {
      supported_groups(10),
      signature_algorithms(13),
      key_share(40),pre_shared_key(41),
      early_data(42),
      supported_versions(43),
      cookie(44),
      psk_key_exchange_modes(45),
      ticket_early_data_info(46),
      (65535)
} ExtensionType;
```

除了NewSessionTicket消息和HelloRetryRequest中的cookie扩展外，服务器发送的扩展只能是客户端扩展的子集，否则客户端会中止握手并发出unsupported_extension警告。客户端发送和服务器同类型但extension_data长度为0的扩展来表示对该扩展类型的支持。

- supported_versions：按偏好顺序排列的支持版本。TLS1.3版本为0x0304。服务器不可发送此类型扩展，因为服务器支持的版本必须写在ServerHello.version（和之前版本的TLS一样）。
- key_share可能包含DH,ECDHE等加密参数。
- psk_key_exchange_modes：主要包括psk_ke，psk_dhe_ke。如果是psk_ke，则服务器禁止提供key_share值。psk_dhe_ke则意味着使用(EC)DHE创建psk。

服务器参数

Encrypted Extensions

和客户端相同。

```
struct {
      Extension extensions<0..2^16-1>;
} EncryptedExtensions;
```

认证请求

```
opaque DistinguishedName<1..2^16-1>;
struct {
opaque certificate_extension_oid<1..2^8-1>;
opaque certificate_extension_values<0..2^16-1>;} CertificateExtension;
struct {
      opaque certificate_request_context<0..2^8-1>;
      SignatureScheme
            supported_signature_algorithms<2..2^16-2>;
      DistinguishedName certificate_authorities<0..2^16-1>;
      CertificateExtension certificate_extensions<0..2^16-1>;
} CertificateRequest;
```

- certificate_request_context：字符串，代表证书请求。

其余字段可以从字面意思理解。

认证信息

在最后的认证阶段，使用如下信息作为输入：

- 证书和使用的签名密钥。
- 基于握手信息哈希值的握手上下文。
- 用于计算MAC的密钥。

基于这些输入，认证信息将包含：

- 证书：当服务器需要认证客户端时需要。包含一个签名算法密钥的证书必须使用不同的签名算法签名。
- 证书验证：这是为了确保握手过程的数据没有被攻击者修改。该数据类型包括签名算法和签名，签名由Hash(Handshake Context + Certificate)得到。
- Finish：标记握手的完成。其数据结构和计算方法如下：

```
finished_key =
HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)

struct {
      opaque verify_data[Hash.length];
} Finished;

verify_data = HMAC(finished_key, Hash(
            Handshake Context +
            Certificate* +
            CertificateVerify*
      )
)
```

握手协议过后的数据传输

服务器收到客户端的结束信息后，可能会发送NewSessionTicket消息。这个消息创建了一个共享密钥。如果需要更改密钥，可以发送Key Update消息。记录层将传输的数据分块，对每一块进行AEAD加密。AEAD的输入由密钥、nonce(sequence number/client_write_iv/server_write_iv)、原始文本、附加数据组成。其中附加数据为空。

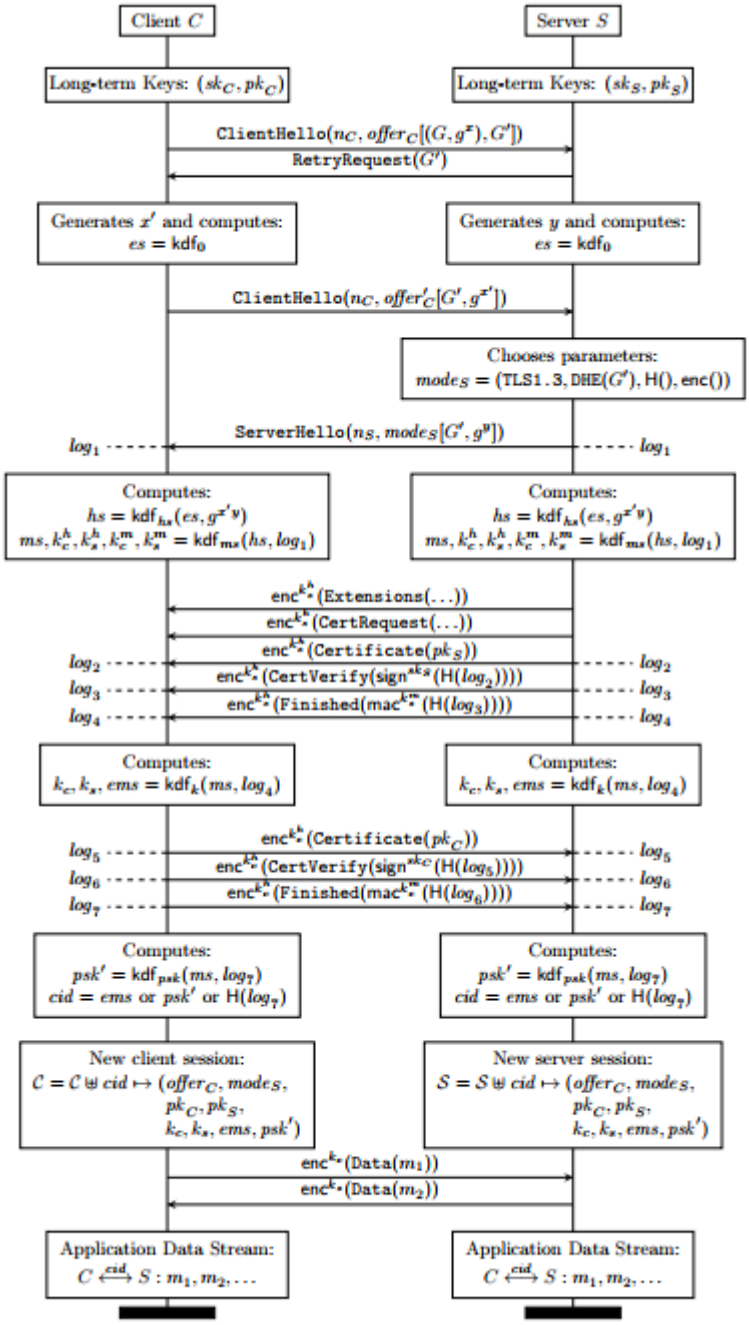
$$AEADEncrypted = AEAD - Encrypt(write\_key, nonce, plaintext\ of\ fragment)$$

$plaintext\ of\ fragment = AEAD - Decrypt(write\_key, nonce, AEADEncrypted)$
64位序列号一开始被初始化为0，每次读或写record都会增加。per-record nonce长度为64位，由以下步骤组成：

- 64为序列号左边iv_length位填0。
- 处理过的序列号和client_write_iv或server_write_iv异或。

使用Proverif验证协议

协议完整流程



其中的key schedule由以下公式导出：
$kdf_0 = hkdf - extract(0^{len_{H()}}, 0^{len_{H()}})$
$kdf_{hs}(es, e) = hkdf - extract(es, e)$
$kdf_{ms}(hs, log_1) = ms, k_c^h, k_s^h, k_c^m, k_s^m\ where$
$ms = hkdf - extract(hs; 0^{len_{H()}})$

$hts_c = derive-secret(hs, hts_c, log_1)$
$hts_s = derive-secret(hs, hts_s, log_1)$
$k_c^h = hkdf-expand-label(hts_c, key, \\')$
$k_c^m = hkdf-expand-label(hts_c, finished, \\')$
$k_s^h = hkdf-expand-label(hts_s, key, \\')$
$k_s^m = hkdf-expand-label(hts_s, finished, \\')$
$kdf_k(ms, log_4) = k_c, k_s, ems\ where$
$ats_c = derive-secret(ms, ats_c, log_4)$
$ats_s = derive-secret(ms, ats_s, log_4)$
$ems = derive-secret(ms, ems, log_4)$
$k_c = hkdf-expand-label(ats_c, key, \\')$
$k_s = hkdf-expand-label(ats_s, key, \\')$
$kdf_{psk}(ms, log_7) = psk'\ where$
$psk' = derive-secret(ms, rms, log_7)$

PSK-based Key Schedule:

$kdf_{es}(psk) = es, k^b\ where$
$es = hkdf-extract(0^{len_{H()}}, psk)$
$k_b = derive-secret(es, pbk, \\')$
$kdf_{0RTT}(es, log_1) = k_c\ where$
$ets_c = derive-secret(es, ets_c, log_1)$
$k_c = hkdf-expand-label(ets_c, key, \\')$

其中H()代表SHA-256，$len_{H()} = 32\ bytes$。

Proverif建模

消息格式和会话状态

以下代码部分新增注释。

```
(*******************************************************)
(*  Message Formats, Session State *)
(*******************************************************)

type random.
type version.
const TLS12, TLS13: version.
type kex_alg.
type rsa_alg.
fun RSA(rsa_alg):kex_alg [data].
fun DHE(group):kex_alg [data].
fun DHE_13(group,element): kex_alg [data].

type psk_type.
fun Binder(bitstring): psk_type [data].
fun NoBinder(): psk_type [data].

type params.
fun nego(version,kex_alg,hash_alg,ae_alg,psk_type): params [data].

type msg.
fun msg2bytes(msg):bitstring [data,typeConverter].
fun CH(random,params):msg [data]. (*Client Hello*)
```

```
fun SH(random,params):msg [data]. (*Server Hello*)
fun CRT(pubkey):msg [data].
fun SKE(group,element,bitstring):msg [data].
fun CKE(bitstring): msg [data].
fun CV(bitstring):msg [data].
fun FIN(bitstring):msg [data]. (*Finish message*)

event ClientOffersVersion(random,version).
event ClientOffersKEX(random,kex_alg).
event ClientOffersAE(random,ae_alg).
event ClientOffersHash(random,hash_alg).

event ClientFinished(version,random,random,
                     preSharedKey,pubkey,
             params,params,
             ae_key,ae_key,bitstring,bitstring).

event ClientFinished0(version,random,preSharedKey,
              params,ae_key,bitstring).

event ServerChoosesVersion(random,random,pubkey,version).
event ServerChoosesKEX(random,random,pubkey,version,kex_alg).
event ServerChoosesAE(random,random,pubkey,version,ae_alg).
event ServerChoosesHash(random,random,pubkey,version,hash_alg).

event ServerFinished0(version,random,preSharedKey,
             params,ae_key).

event ServerFinished(version,random,random,
                     preSharedKey,pubkey,
             params,params,
             ae_key,ae_key,bitstring,bitstring).
event PreServerFinished(version,random,random,
            preSharedKey,pubkey,
            params,params,
            ae_key,ae_key,bitstring).

table clientSession(random,random,preSharedKey,pubkey,
            params,params,
            ae_key,ae_key,bitstring,bitstring).
table serverSession(random,random,preSharedKey,pubkey,
            params,params,
            ae_key,ae_key,bitstring,bitstring).

table serverSession0_5(random,random,preSharedKey,pubkey,
            params,params,
            ae_key,ae_key,bitstring).

table clientSession0(random,preSharedKey,params,ae_key,bitstring).
table serverSession0(random,preSharedKey,params,ae_key,bitstring).
```

各部分含义较明显，其中clientSession0保存的是Hello问候结束后创建的会话，serverSession0同理。

客户端和服务器处理流程

首先客户端发出ClientHello(简记为CH):

```
let Client13() =
    (get preSharedKeys(a,b,psk) in
     in (io,ioffer:params);
     let nego(=TLS13,DHE_13(g,eee),hhh,aaa,pt) = ioffer in
     new cr:random;
     let (x:bitstring,gx:element) = dh_keygen(g) in
     let (early_secret:bitstring,kb:mac_key) = kdf_es(psk) in
     let zoffer = nego(TLS13,DHE_13(g,gx),hhh,aaa,Binder(zero)) in
     let pt = Binder(hmac(StrongHash,kb,msg2bytes(CH(cr,zoffer)))) in
     let offer = nego(TLS13,DHE_13(g,gx),hhh,aaa,pt) in
     let ch = CH(cr,offer) in
     event ClientOffersVersion(cr,TLS13);
     event ClientOffersKEX(cr,DHE_13(g,gx));
     event ClientOffersAE(cr,aaa);
     event ClientOffersHash(cr,hhh);
     out(io,ch);
     let (kc0:ae_key,ems0:bitstring) = kdf_k0(early_secret,msg2bytes(ch)) in
     insert clientSession0(cr,psk,offer,kc0,ems0);
     ...(omitted)
```

ch包含了客户端生成的随机数、共享密钥的一部分等。
服务器收到ch，发送ServerHello(SH):

```
let Server13() =
    (get preSharedKeys(a,b,psk) in
     in(io,ch:msg);
     let CH(cr,offer) = ch in
     let nego(=TLS13,DHE_13(g,gx),hhh,aaa,Binder(m)) = offer in
     let (early_secret:bitstring,kb:mac_key) = kdf_es(psk) in
     let zoffer = nego(TLS13,DHE_13(g,gx),hhh,aaa,Binder(zero)) in
     if m = hmac(StrongHash,kb,msg2bytes(CH(cr,zoffer))) then
     let (kc0:ae_key,ems0:bitstring) =
         kdf_k0(early_secret,msg2bytes(ch)) in
     insert serverSession0(cr,psk,offer,kc0,ems0);

     new sr:random;
     in(io,SH(xxx,mode));
     let nego(=TLS13,DHE_13(=g,eee),h,a,pt) = mode in
     let (y:bitstring,gy:element) = dh_keygen(g) in
     let mode = nego(TLS13,DHE_13(g,gy),h,a,pt) in
     out(io,SH(sr,mode));
     ...omitted
```

客户端收到mode，将其解析，随后生成密钥信息发给服务器:

```
in(io,SH(sr,mode));
     let nego(=TLS13,DHE_13(=g,gy),h,a,spt) = mode in
     let log = (ch,SH(sr,mode)) in

     let gxy = e2b(dh_exp(g,gy,x)) in
     let handshake_secret = kdf_hs(early_secret,gxy) in
     let (master_secret:bitstring,chk:ae_key,shk:ae_key,cfin:mac_key,sfin:mac_key)
=
         kdf_ms(handshake_secret,log) in
     out(io,(chk,shk));
```

服务器也类似：

```
let log = (ch,SH(sr,mode)) in
     get longTermKeys_tbl(sn,sk,p) in
     event ServerChoosesVersion(cr,sr,p,TLS13);
     event ServerChoosesKEX(cr,sr,p,TLS13,DHE_13(g,gy));
     event ServerChoosesAE(cr,sr,p,TLS13,a);
     event ServerChoosesHash(cr,sr,p,TLS13,h);

     let gxy = e2b(dh_exp(g,gx,y)) in
     let handshake_secret = kdf_hs(early_secret,gxy) in
     let (master_secret:bitstring,chk:ae_key,shk:ae_key,cfin:mac_key,sfin:mac_key)
=
         kdf_ms(handshake_secret,log) in
     out(io,(chk,shk));
```

服务器开始认证，认证结束后发送Finish：

```
out(io,CRT(p));
     let log = (log,CRT(p)) in
     let sg = sign(sk,hash(h,log)) in
     out(io,CV(sg));
     let log = (log,CV(sg)) in
     let m1 = hmac(StrongHash,sfin,log) in

     let log = (log,FIN(m1)) in

     let (cak:ae_key,sak:ae_key,ems:bitstring) = kdf_k(master_secret,log) in
     event PreServerFinished(TLS13,cr,sr,psk,p,offer,mode,cak,sak,ems);
     out(io,FIN(m1));
```

客户端收到服务器的认证消息进行认证，最后也发送Finish：

```
in(io,CRT(p)); (*receive pks*)
     let log = (log,CRT(p)) in
     get longTermKeys_tbl(sn,xxx,=p) in
     in(io,CV(s)); (*s=sign(sk,hash(h,log))*)
     if verify(p,hash(h,log),s) = true then
     let log = (log,CV(s)) in
     in(io,FIN(m1));
     if m1 = hmac(StrongHash,sfin,log) then (
        let log = (log,FIN(m1)) in
```

```
    let (cak:ae_key,sak:ae_key,ems:bitstring) = kdf_k(master_secret,log) in
        let m2 = hmac(StrongHash,cfin,log) in
        let log = (log,FIN(m2)) in
    let rms =  kdf_psk(master_secret,log) in

        event ClientFinished(TLS13,cr,sr,psk,p,offer,mode,cak,sak,ems,rms);
        insert clientSession(cr,sr,psk,p,offer,mode,cak,sak,ems,rms);
        out(io,FIN(m2))))).
```

服务器收到后发出结束事件并创建信的会话:

```
in(io,FIN(m2));
     if m2 = hmac(StrongHash,cfin,log) then
        let log = (log,FIN(m2)) in
    let rms =  kdf_psk(master_secret,log) in
        event ServerFinished(TLS13,cr,sr,psk,p,offer,mode,cak,sak,ems,rms);
        insert serverSession(cr,sr,psk,p,offer,mode,cak,sak,ems,rms);
        phase 1;
        event PostSessionCompromisedKey(pk(sk));
    out(io,sk)).
```

appData模拟应用层数据的传输，传输方进行AEAD编码，接受方进行AEAD解码，随后响应信息给传输方。

```
(*********************************************************)
(*  Application Data Client and Server (+Record Layer) *)
(*********************************************************)

event ClientSends0(version,random,preSharedKey,bitstring,bitstring,bitstring).
event ServerReceives0(version,random,preSharedKey,bitstring,bitstring,bitstring).
event
ClientSends(version,random,random,preSharedKey,pubkey,bitstring,bitstring,bitstring
).
event
ServerSends(version,random,random,preSharedKey,pubkey,bitstring,bitstring,bitstring
).
event
ClientReceives(version,random,random,preSharedKey,pubkey,bitstring,bitstring,bitstr
ing).
event
ServerReceives(version,random,random,preSharedKey,pubkey,bitstring,bitstring,bitstr
ing).


fun m_c0(version,random,preSharedKey): bitstring [private].
fun m_s(version,random,random,pubkey,preSharedKey): bitstring [private].
fun m_c(version,random,random,pubkey,preSharedKey): bitstring [private].

let appData() =
    (get clientSession0(cr,psk,o,kc0,ems0) in
     let nego(v,kkk,hhh,a,pt) = o in
     in (io,(n:bitstring, ad:bitstring));
     let mesg = m_c0(TLS13,cr,psk) in
     event ClientSends0(TLS13,cr,psk,n,ad,mesg);
```

```
     out (io,aead_enc(a,kc0,n,ad,mesg))) (*fun aead_enc(ae_alg, ae_key, bitstring,
bitstring, bitstring): bitstring.*)
  |
    (get serverSession0(cr,psk,o,kc0,ems0) in
     let nego(v,kkk,hhh,a,pt) = o in
     in (io,(n:bitstring, ad:bitstring, c:bitstring));
     let f = aead_dec(a,kc0,n,ad,c) in
     event ServerReceives0(TLS13,cr,psk,n,ad,f))
  |
    (get serverSession0_5(cr,sr,psk,ps,o,m,kc,ks,ems) in
     let nego(v,kkk,hhh,a,pt) = m in
     in (io,(n:bitstring, ad:bitstring));
     let mesg = m_s(v,cr,sr,ps,psk) in
     event ServerSends(v,cr,sr,psk,ps,n,ad,mesg);
     out (io,aead_enc(a,ks,n,ad,mesg)))
  |
    (get clientSession(cr,sr,psk,ps,o,m,kc,ks,ems,rms) in
     let nego(v,kkk,hhh,a,pt) = m in
     in (io,(n:bitstring, ad:bitstring));
     let mesg = m_c(v,cr,sr,ps,psk) in
     event ClientSends(v,cr,sr,psk,ps,n,ad,mesg);
     out (io,aead_enc(a,kc,n,ad,mesg)))
  |
    (get serverSession(cr,sr,psk,ps,o,m,kc,ks,ems,rms) in
     let nego(v,kkk,hhh,a,pt) = m in
     in (io,(n:bitstring, ad:bitstring, c:bitstring));
     let f = aead_dec(a,kc,n,ad,c) in
     event ServerReceives(v,cr,sr,psk,ps,n,ad,f))
  |
    (get serverSession(cr,sr,psk,ps,o,m,kc,ks,ems,rms) in
     let nego(v,kkk,hhh,a,pt) = m in
     in (io,(n:bitstring, ad:bitstring));
     let mesg = m_s(v,cr,sr,ps,psk) in
     event ServerSends(v,cr,sr,psk,ps,n,ad,mesg);
     out (io,aead_enc(a,ks,n,ad,mesg)))
  |
    (get clientSession(cr,sr,psk,ps,o,m,kc,ks,ems,rms) in
     let nego(v,kkk,hhh,a,pt) = m in
     in (io,(n:bitstring, ad:bitstring, c:bitstring));
     let f = aead_dec(a,ks,n,ad,c) in
     event ClientReceives(v,cr,sr,psk,ps,n,ad,f))
.
```

几类安全查询

查询什么时候密钥泄漏：

```
event ClientAEKeyLeaked(version,random,random,preSharedKey,pubkey).
event ServerAEKeyLeaked(version,random,random,preSharedKey,pubkey).
event ClientAEKeyLeaked0(version,random,preSharedKey,params).
event ServerAEKeyLeaked0(version,random,preSharedKey,params).

let secrecyQuery() =
    (get clientSession(cr,sr,psk,p,o,m,ck,sk,cb,ms) in
```

```
     let nego(v,kkk,hhh,aaa,ppp) = m in
     in (io,=ck);
     event ClientAEKeyLeaked(v,cr,sr,psk,p))
 |  (get serverSession(cr,sr,psk,p,o,m,ck,sk,cb,ms) in
     let nego(v,kkk,hhh,aaa,ppp) = m in
     in (io,=sk);
     event ServerAEKeyLeaked(v,cr,sr,psk,p))
 |  (get clientSession0(cr,psk,o,ck,ems) in
     in (io,=ck);
     event ClientAEKeyLeaked0(TLS13,cr,psk,o))
 |  (get serverSession0(cr,psk,o,ck,ems) in
     in (io,=ck);
     event ServerAEKeyLeaked0(TLS13,cr,psk,o)).
```

会话参数的唯一性：

有一些会话参数，一旦相等必须说明同属于一个会话。考虑它的反面，即这个参数相等但是其他参数不完全相同。测试结果应该全为False。

```
event
MatchingChannelBinding(version,random,random,pubkey,version,random,random,pubkey).
event
MatchingResumptionSecret(version,random,random,pubkey,version,random,random,pubkey)
.
event MatchingAEKey(version,random,random,pubkey,version,random,random,pubkey).
event
MatchingAEKey0(version,random,preSharedKey,params,random,preSharedKey,params).
event MatchingEMS0(version,random,preSharedKey,params,random,preSharedKey,params).

let channelBindingQuery() =
    (get clientSession0(cr,psk,o,ck,ems) in
     get serverSession0(cr',psk',o',=ck,ems) in
     if (cr <> cr' || psk <> psk' || o <> o') then
     event MatchingAEKey0(TLS13,cr,psk,o,cr',psk',o'))
|   (get clientSession0(cr,psk,o,ck,ems) in
     get serverSession0(cr',psk',o',ck,=ems) in
     if (cr <> cr' || psk <> psk' || o <> o') then
     event MatchingEMS0(TLS13,cr,psk,o,cr',psk',o'))
|   (get clientSession(cr,sr,psk,p,o,m,ck,sk,cb,ms) in
     get serverSession(cr',sr',psk',p',o',m',ck',sk',=cb,ms') in
     let nego(v,kkk,hhh,aaa,ppp) = m in
     let nego(v',kkk',hhh',aaa',ppp') = m' in
     if (cr <> cr' || sr <> sr' || p <> p') then
     event MatchingChannelBinding(v,cr,sr,p,v',cr',sr',p'))
|   (get clientSession(cr,sr,psk,p,o,m,ck,sk,cb,ms) in
     get serverSession(cr',sr',psk',p',o',m',ck',sk',cb',=ms) in
     let nego(v,kkk,hhh,aaa,ppp) = m in
     let nego(v',kkk',hhh',aaa',ppp') = m' in
     if (cr <> cr' || sr <> sr' || p <> p') then
     event MatchingResumptionSecret(v,cr,sr,p,v',cr',sr',p'))
|   (get clientSession(cr,sr,psk,p,o,m,ck,sk,cb,ms) in
     get serverSession(cr',sr',psk',p',o',m',=ck,=sk,cb',ms') in
     let nego(v,kkk,hhh,aaa,ppp) = m in
     let nego(v',kkk',hhh',aaa',ppp') = m' in
```

```
     if (cr <> cr' || sr <> sr' || p <> p') then
     event MatchingAEKey(v,cr,sr,p,v',cr',sr',p')).
```

有关认证顺序：

应该是先服务器发出Finish再客户端Finish。除此之外还有一些其他结束顺序相关：

```
(*********************************************************)
(*  Sanity Queries: should all be false  *)
(*********************************************************)

query cr:random, sr:random,
      psk:preSharedKey,p:pubkey,o:params, m:params,
      ck:ae_key, sk:ae_key, cb:bitstring, ms:bitstring;
      event(ClientFinished(TLS12,cr,sr,psk,p,m,o,ck,sk,cb,ms)) ==>
      event(ServerFinished(TLS12,cr,sr,psk,p,m,o,ck,sk,cb,ms)).

query cr:random, sr:random,
      psk:preSharedKey,p:pubkey,o:params, m:params,
      ck:ae_key, sk:ae_key, cb:bitstring, ms:bitstring;
      event(ClientFinished(TLS13,cr,sr,psk,p,m,o,ck,sk,cb,ms)) ==>
      event(PreServerFinished(TLS13,cr,sr,psk,p,m,o,ck,sk,cb)).

query cr:random, sr:random,
      psk:preSharedKey,p:pubkey,o:params, m:params,
      ck:ae_key, sk:ae_key, cb:bitstring, ms:bitstring;
      event(ServerFinished(TLS12,cr,sr,psk,p,m,o,ck,sk,cb,ms)).

query cr:random, sr:random,
      psk:preSharedKey,p:pubkey,o:params, m:params,
      ck:ae_key, sk:ae_key, cb:bitstring, ms:bitstring;
      event(ClientFinished(TLS12,cr,sr,psk,p,m,o,ck,sk,cb,ms)).

query cr:random, sr:random,
      psk:preSharedKey,p:pubkey,o:params, m:params,
      ck:ae_key, sk:ae_key, cb:bitstring, ms:bitstring;
      event(ServerFinished(TLS13,cr,sr,psk,p,m,o,ck,sk,cb,ms)).

query cr:random, sr:random,
      psk:preSharedKey,p:pubkey,o:params, m:params,
      ck:ae_key, sk:ae_key, cb:bitstring, ms:bitstring;
      event(PreServerFinished(TLS13,cr,sr,psk,p,m,o,ck,sk,cb)).

query cr:random, sr:random,
      psk:preSharedKey,p:pubkey,o:params, m:params,
      ck:ae_key, sk:ae_key, cb:bitstring, ms:bitstring;
      event(ClientFinished(TLS13,cr,sr,psk,p,m,o,ck,sk,cb,ms)).
```

有关弱算法导致密钥泄漏：

```
(* Following secrecy queries should fail *)
query cr:random, sr:random, psk:preSharedKey, p:pubkey, ms:bitstring, aek:ae_key,
cb:bitstring, cr':random, sr':random, v:version, e:element;
      event(ClientAEKeyLeaked(TLS13,cr,sr,psk,p)) ==>
```

```
      (event(WeakOrCompromisedKey(p)) && (psk = NoPSK ||
event(CompromisedPreSharedKey(psk)))) ||
      event(ServerChoosesKEX(cr,sr,p,TLS13,DHE_13(WeakDH,e))).

query cr:random, sr:random, psk:preSharedKey, p:pubkey, ms:bitstring, aek:ae_key,
cb:bitstring, cr':random, sr':random, v:version, e:element;
      event(ClientAEKeyLeaked(TLS13,cr,sr,psk,p)) ==>
      (event(WeakOrCompromisedKey(p)) && (psk = NoPSK ||
event(CompromisedPreSharedKey(psk)))) ||
      event(ServerChoosesHash(cr',sr',p,TLS13,WeakHash)).

query cr:random, sr:random, psk:preSharedKey, p:pubkey, ms:bitstring, aek:ae_key,
cb:bitstring, cr':random, sr':random, v:version, e:element;
      event(ClientAEKeyLeaked(TLS13,cr,sr,psk,p)) ==>
      event(ServerChoosesKEX(cr,sr,p,TLS13,DHE_13(WeakDH,e))) ||
      event(ServerChoosesHash(cr',sr',p,TLS13,WeakHash)).
```

其余验证点从略（可以从稍后的验证结果中找出）。

验证结果

```
Verification summary:

Query
event(ClientFinished(TLS12,cr_18,sr_12,psk_18,p_7,m_13,o_16,ck_10,sk_6,cb_5,ms_7))
==>
event(ServerFinished(TLS12,cr_18,sr_12,psk_18,p_7,m_13,o_16,ck_10,sk_6,cb_5,ms_7))
is true.

Query
event(ClientFinished(TLS13,cr_18,sr_12,psk_18,p_7,m_13,o_16,ck_10,sk_6,cb_5,ms_7))
==>
event(PreServerFinished(TLS13,cr_18,sr_12,psk_18,p_7,m_13,o_16,ck_10,sk_6,cb_5)) is
false.

Query not
event(ServerFinished(TLS12,cr_18,sr_12,psk_18,p_7,m_13,o_16,ck_10,sk_6,cb_5,ms_7))
is true.

Query not
event(ClientFinished(TLS12,cr_18,sr_12,psk_18,p_7,m_13,o_16,ck_10,sk_6,cb_5,ms_7))
is true.

Query not
event(ServerFinished(TLS13,cr_18,sr_12,psk_18,p_7,m_13,o_16,ck_10,sk_6,cb_5,ms_7))
is false.

Query not
event(PreServerFinished(TLS13,cr_18,sr_12,psk_18,p_7,m_13,o_16,ck_10,sk_6,cb_5)) is
false.

Query not
event(ClientFinished(TLS13,cr_18,sr_12,psk_18,p_7,m_13,o_16,ck_10,sk_6,cb_5,ms_7))
is false.
```

```
Query event(ClientAEKeyLeaked(TLS13,cr_18,sr_12,psk_18,p_7)) ==>
(event(WeakOrCompromisedKey(p_7)) && (psk_18 = NoPSK ||
event(CompromisedPreSharedKey(psk_18)))) ||
event(ServerChoosesKEX(cr_18,sr_12,p_7,TLS13,DHE_13(WeakDH,e))) is false.

Query event(ClientAEKeyLeaked(TLS13,cr_18,sr_12,psk_18,p_7)) ==>
(event(WeakOrCompromisedKey(p_7)) && (psk_18 = NoPSK ||
event(CompromisedPreSharedKey(psk_18)))) ||
event(ServerChoosesHash(cr'_5,sr'_3,p_7,TLS13,WeakHash)) is false.

Query event(ClientAEKeyLeaked(TLS13,cr_18,sr_12,psk_18,p_7)) ==>
event(ServerChoosesKEX(cr_18,sr_12,p_7,TLS13,DHE_13(WeakDH,e))) ||
event(ServerChoosesHash(cr'_5,sr'_3,p_7,TLS13,WeakHash)) is false.

Query inj-
event(ClientFinished(TLS13,cr_18,sr_12,psk_18,p_7,o_16,m_13,ck_10,sk_6,cb_5,ms_7))
==> inj-
event(PreServerFinished(TLS13,cr_18,sr_12,psk_18,p_7,o_16,m_13,ck_10,sk_6,cb_5)) ||
(event(WeakOrCompromisedKey(p_7)) && (psk_18 = NoPSK ||
event(CompromisedPreSharedKey(psk_18)))) is false.

Query inj-
event(ClientFinished(TLS13,cr_18,sr_12,psk_18,p_7,o_16,m_13,ck_10,sk_6,cb_5,ms_7))
==> inj-
event(PreServerFinished(TLS13,cr_18,sr_12,psk_18,p_7,o_16,m_13,ck_10,sk_6,cb_5)) ||
(event(WeakOrCompromisedKey(p_7)) && (psk_18 = NoPSK ||
event(CompromisedPreSharedKey(psk_18)))) ||
event(ServerChoosesKEX(cr_18,sr_12,p_7,TLS13,DHE_13(WeakDH,e))) is false.

Query inj-
event(ClientFinished(TLS13,cr_18,sr_12,psk_18,p_7,o_16,m_13,ck_10,sk_6,cb_5,ms_7))
==> inj-
event(PreServerFinished(TLS13,cr_18,sr_12,psk_18,p_7,o_16,m_13,ck_10,sk_6,cb_5)) ||
event(ServerChoosesKEX(cr_18,sr_12,p_7,TLS13,DHE_13(WeakDH,e))) ||
event(ServerChoosesHash(cr'_5,sr'_3,p_7,TLS13,WeakHash)) is false.

Query inj-
event(ClientFinished(TLS13,cr_18,sr_12,psk_18,p_7,o_16,m_13,ck_10,sk_6,cb_5,ms_7))
==> inj-
event(PreServerFinished(TLS13,cr_18,sr_12,psk_18,p_7,o_16,m_13,ck_10,sk_6,cb_5)) is
false.

Query not
event(MatchingChannelBinding(TLS13,cr_18,sr_12,p_7,TLS13,cr'_5,sr'_3,p'_3)) is
true.

Query not
event(MatchingResumptionSecret(TLS13,cr_18,sr_12,p_7,TLS13,cr'_5,sr'_3,p'_3)) is
true.

Query not event(MatchingAEKey(TLS13,cr_18,sr_12,p_7,TLS13,cr'_5,sr'_3,p'_3)) is
true.
```

```
Query event(ClientAEKeyLeaked(TLS13,cr_18,sr_12,psk_18,p_7)) ==>
(event(WeakOrCompromisedKey(p_7)) && (psk_18 = NoPSK ||
event(CompromisedPreSharedKey(psk_18)))) ||
event(ServerChoosesKEX(cr_18,sr_12,p_7,TLS13,DHE_13(WeakDH,e))) ||
event(ServerChoosesHash(cr'_5,sr'_3,p_7,TLS13,WeakHash)) is true.

Query inj-
event(ClientFinished(TLS13,cr_18,sr_12,psk_18,p_7,o_16,m_13,ck_10,sk_6,cb_5,ms_7))
==> inj-
event(PreServerFinished(TLS13,cr_18,sr_12,psk_18,p_7,o_16,m_13,ck_10,sk_6,cb_5)) ||
(event(WeakOrCompromisedKey(p_7)) && (psk_18 = NoPSK ||
event(CompromisedPreSharedKey(psk_18)))) ||
event(ServerChoosesKEX(cr_18,sr_12,p_7,TLS13,DHE_13(WeakDH,e))) ||
event(ServerChoosesHash(cr'_5,sr'_3,p_7,TLS13,WeakHash)) is true.

Query attacker_p1(m_c(TLS13,cr_18,sr_12,p_7,psk_18)) ==>
(event(WeakOrCompromisedKey(p_7)) && (psk_18 = NoPSK ||
event(CompromisedPreSharedKey(psk_18)))) ||
event(ServerChoosesAE(cr_18,sr_12,p_7,TLS13,WeakAE)) ||
event(ServerChoosesKEX(cr_18,sr_12,p_7,TLS13,DHE_13(WeakDH,e))) ||
event(ServerChoosesHash(cr'_5,sr'_3,p_7,TLS13,WeakHash)) is true.

Query event(ClientReceives(TLS13,cr_18,sr_12,psk_18,p_7,n_7,ad_7,m_13)) ==>
event(ServerSends(TLS13,cr_18,sr_12,psk_18,p_7,n_7,ad_7,m_13)) ||
(event(WeakOrCompromisedKey(p_7)) && (psk_18 = NoPSK ||
event(CompromisedPreSharedKey(psk_18)))) ||
event(ServerChoosesAE(cr_18,sr_12,p_7,TLS13,WeakAE)) ||
event(ServerChoosesKEX(cr_18,sr_12,p_7,TLS13,DHE_13(WeakDH,e))) ||
event(ServerChoosesHash(cr'_5,sr'_3,p_7,TLS13,WeakHash)) is true.

Query event(ServerReceives(TLS13,cr_18,sr_12,psk_18,p_7,n_7,ad_7,m_13)) ==>
event(ClientSends(TLS13,cr_18,sr_12,psk_18,p_7,n_7,ad_7,m_13)) || psk_18 = NoPSK ||
event(CompromisedPreSharedKey(psk_18)) ||
event(ServerChoosesAE(cr_18,sr_12,p_7,TLS13,WeakAE)) ||
event(ServerChoosesKEX(cr_18,sr_12,p_7,TLS13,DHE_13(WeakDH,e))) ||
event(ServerChoosesHash(cr'_5,sr'_3,p_7,TLS13,WeakHash)) is true.

Query event(ServerReceives0(TLS13,cr_18,psk_18,n_7,ad_7,m_13)) ==>
event(ClientSends0(TLS13,cr_18,psk_18,n_7,ad_7,m_13)) || psk_18 = NoPSK ||
event(CompromisedPreSharedKey(psk_18)) || event(ClientOffersAE(cr_18,WeakAE)) is
true.
```

通过前面几次查询对TLS 1.2和TLS 1.3的对比，发现TLS 1.3的安全性确实更强。除此之外，举一点进行详细说明：

```
Query event(ClientAEKeyLeaked(TLS13,cr_18,sr_12,psk_18,p_7)) ==>
event(ServerChoosesKEX(cr_18,sr_12,p_7,TLS13,DHE_13(WeakDH,e))) ||
event(ServerChoosesHash(cr'_5,sr'_3,p_7,TLS13,WeakHash)) is false.
```

说明在选择WeakDH和WeakHash的情况下并不会造成AE密钥泄露。当然，在多种配置不当的条件下，还是会导致数据被窃取：

```
Query attacker_p1(m_c(TLS13,cr_18,sr_12,p_7,psk_18)) ==>
(event(WeakOrCompromisedKey(p_7)) && (psk_18 = NoPSK ||
event(CompromisedPreSharedKey(psk_18)))) ||
event(ServerChoosesAE(cr_18,sr_12,p_7,TLS13,WeakAE)) ||
event(ServerChoosesKEX(cr_18,sr_12,p_7,TLS13,DHE_13(WeakDH,e))) ||
event(ServerChoosesHash(cr'_5,sr'_3,p_7,TLS13,WeakHash)) is true.
```

其余结果可以类似分析。

总结

- 相对于TLS 1.2，TLS 1.3在安全性方面做了很大改进，比如不会因为客户端先发出Finished信号而导致认证未完成。TLS 1.3很好地通过了本文中对安全性能的测试。
- 本文详细讲述了TLS 1.3协议，并使用自动化工具认证了它的安全性。阅读本文主要的难点是对TLS协议和对应Proverif形式化语言的理解。

参考资料

- https://datatracker.ietf.org/doc/pdf/rfc5116 An Interface and Algorithms for Authenticated Encryption
- https://datatracker.ietf.org/doc/pdf/rfc5869 HMAC-based Extract-and-Expand Key Derivation Function (HKDF)
- https://www.cnblogs.com/foxclever/p/8642865.html 信息摘要算法之六：HKDF算法分析与实现
- https://bblanche.gitlabpages.inria.fr/publications/BhargavanBlanchetKobeissiSP2017.html Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate
- https://theswissbay.ch/pdf/Books/Computer%20science/Cryptography/SeriousCryptography.pdf Serious Cryptography——A Practical Introduction
  A Practical Introduction to Modern Encryption to Modern Encryption
- https://en.wikipedia.org/wiki/Authenticated_encryption Authenticated encryption
- https://en.wikipedia.org/wiki/HMAC HMAC
- https://zhuanlan.zhihu.com/p/74994669 Diffie-Hellman密钥交换协议