Le protocole TLS/SSL

Ses versions, des attaques, TLS 1.3

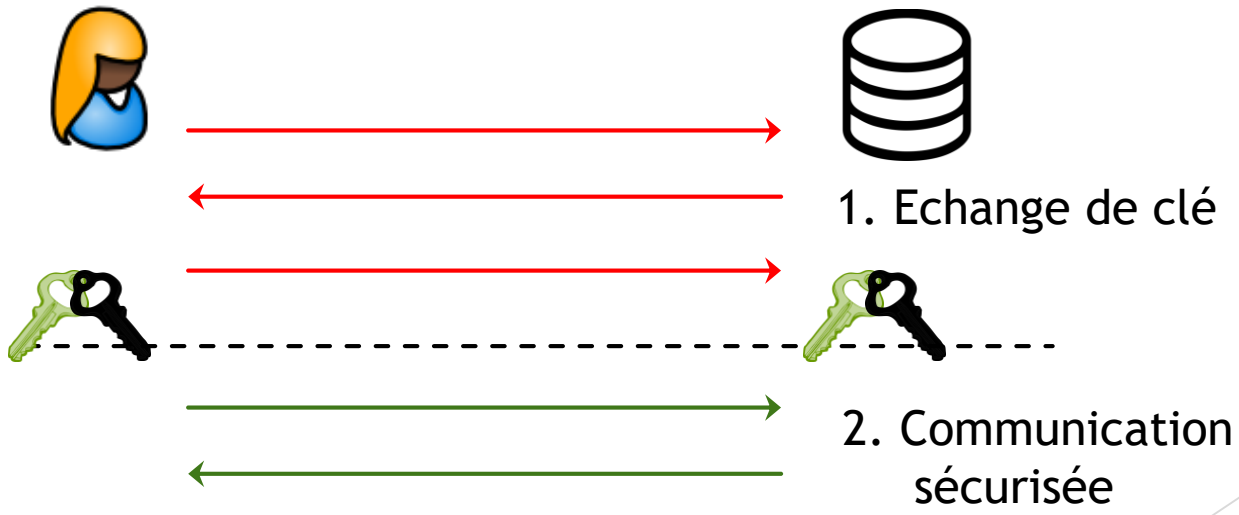
Cristina ONETE

maria-cristina.onete@unilim.fr

La notion de canal sécurisé

- Alice veut se connecter à sa banque en ligne
  - Elle veut s'assurer qu'elle parle vraiment à sa banque
  - ... et que personne d'autre ne pourra pas voir le contenu de la communication

Alice et sa banque vont établir et utiliser un canal sécurisé



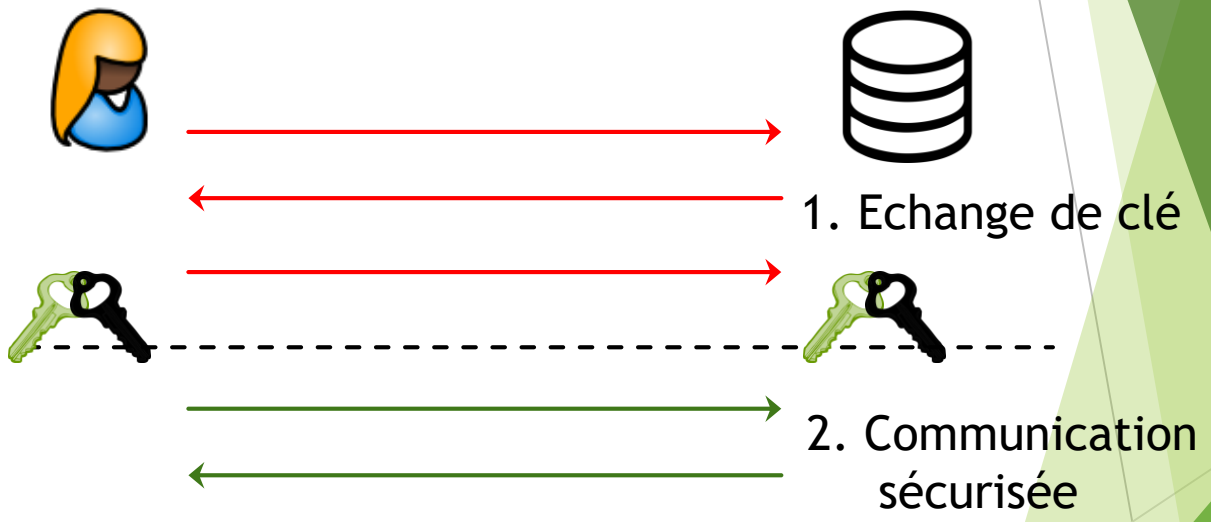
Les canaux sécurisés en pratique

- Trois protocoles majeurs utilisés en pratique :
  - **IPSec** : suite de protocoles développés pour sécuriser une connexion sur VPN
  - **SSH** : protocole utilisé pour la plupart pour sécuriser un accès à distance à un autre ordinateur (dans un shell)
  - **TLS** : protocole utilisé pour sécuriser la navigation sur l'Internet, mais aussi pour sécuriser la connexion entre deux composants d'un reseaux mobile, etc.

Rappel : Notions sur l'échange de clé & le chiffrement authentifié

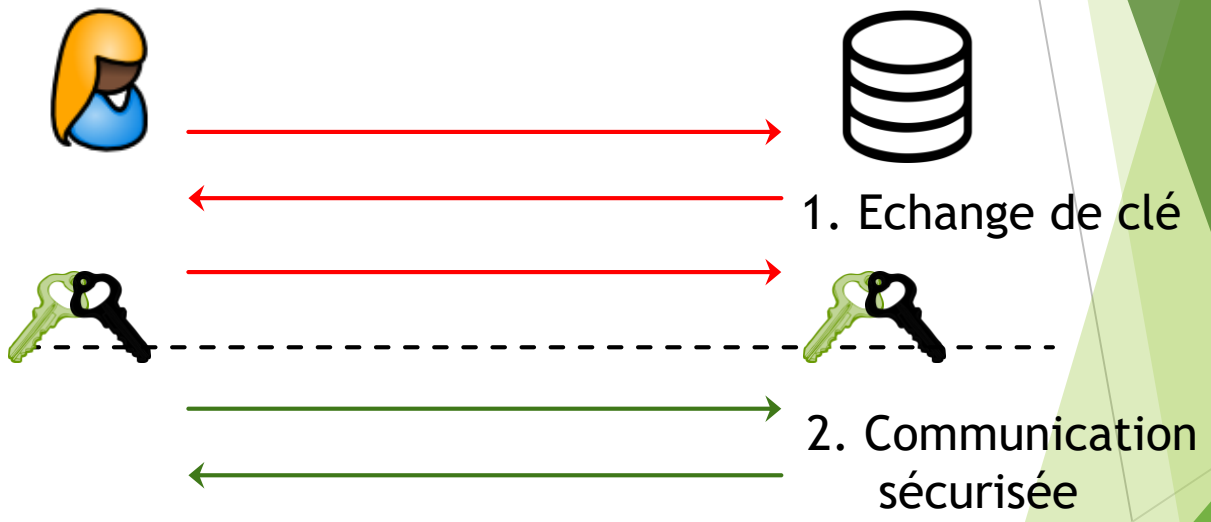
Eléments principaux

- 3 éléments principaux :
  - L'échange de clé authentifié
  - La dérivation des clés
  - L'utilisation de clés pour une communication sécurisée



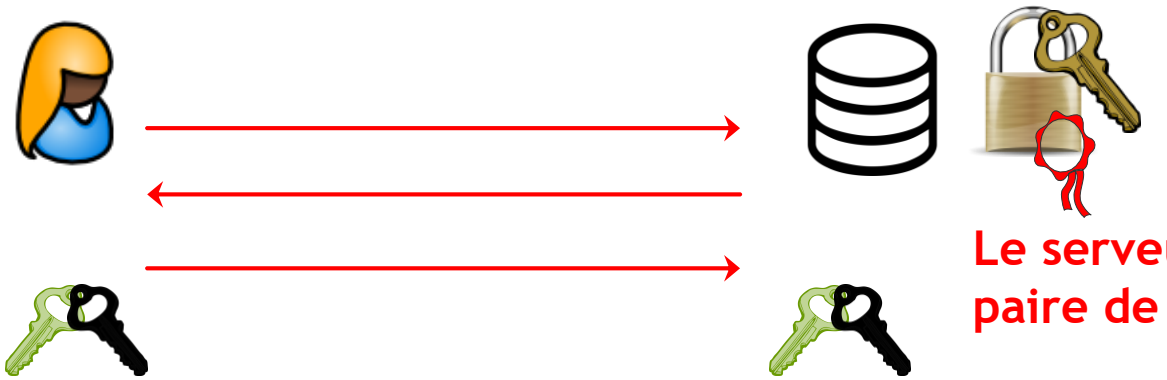
Eléments principaux

- 3 éléments prinxipaux :
  - **L'échange de clé authentifié**
  - La dérivation des clés
  - L'utilisation de clés pour une communication sécurisée



Echange de clé authentifié

- En pratique, l'authentification est **unilaterale**
- Alice sera certaine qu'elle parle à sa banque
- Sa banque aura besoin plus tard de son login + mot de passe
- Echange de messages en claire pour calculer des clés de sessions (confidentielles)

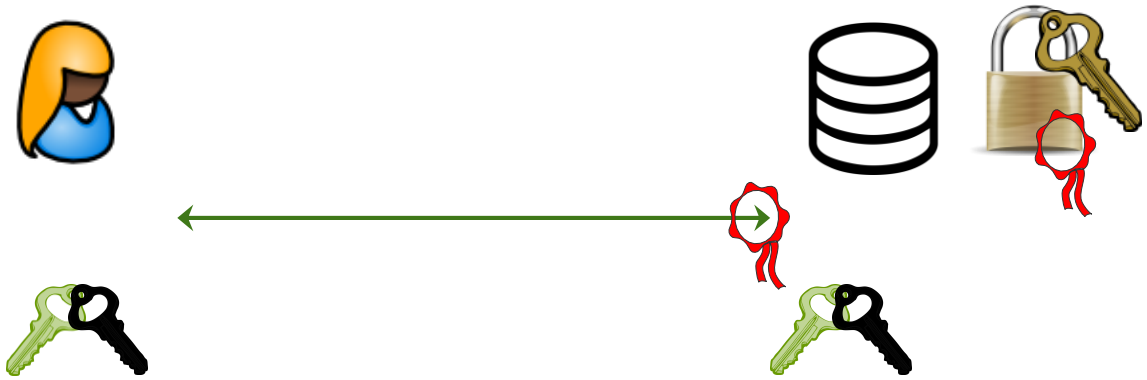


Le serveur a une paire de clés certifiées

Alice authentifie le serveur avant le calcul de clés de session

Sécurité de la clé

- En pratique, l'authentification est **unilaterale**
- Echange de messages en claire pour calculer des clés de sessions (confidentielles)



Sécurité : -- Un attaquant **peut ouvrir une session** avec la banque
-- Mais **si Alice en ouvre une**, l'attaquant n'aura **aucune information** sur les clés calculées

PFS : -- Si un attaquant apprend la clé secrète du serveur, alors aucune des sessions passées n'est affectée

Rappel: Diffie Hellman

- Alice et Bob veulent échanger une clé
- On utilise un corps fini, par exemple $\mathrm{G} = < g >$, d'ordre premier q

Choisir $a \in \{1, \dots, q-1\}$

Choisir $b \in \{1, \dots, q-1\}$

g^a →

← g^b

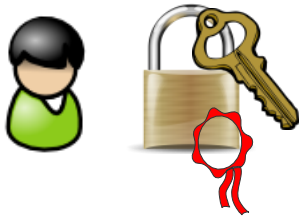
Calcule $K = (g^b)^a$

Calcule $K = (g^a)^b$

Seulement sécurisé contre des attaquants passifs

Echange de clé avec RSA (KEM)

- On suppose que Bob a une paire de clés de chiffrement RSA (sk_B, pk_B)



Choisir K aléatoirement

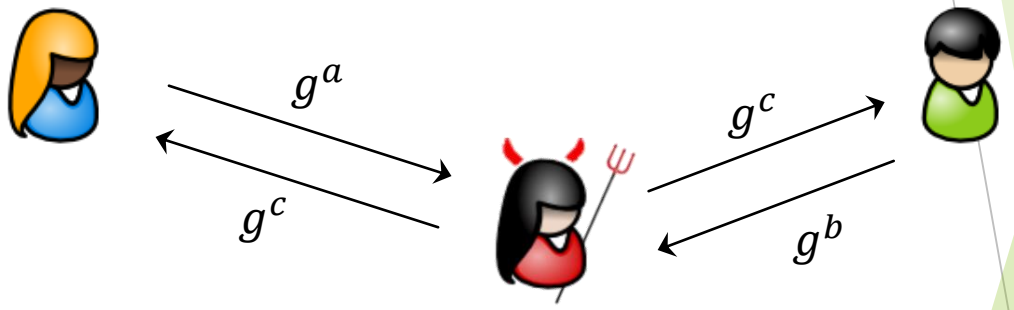
$c = \mathrm{RSA.Enc}(pk_B; K)$

(sk_B, pk_B)

$K = \mathrm{RSA.Dec}(sk_B; c)$

Diffie Hellman -- un problème

- Aucun moyen de s'assurer de l'authenticité de données
  - Clé à gauche $K = g^{ac}$
  - Clé à droite $K^* = g^{bc}$

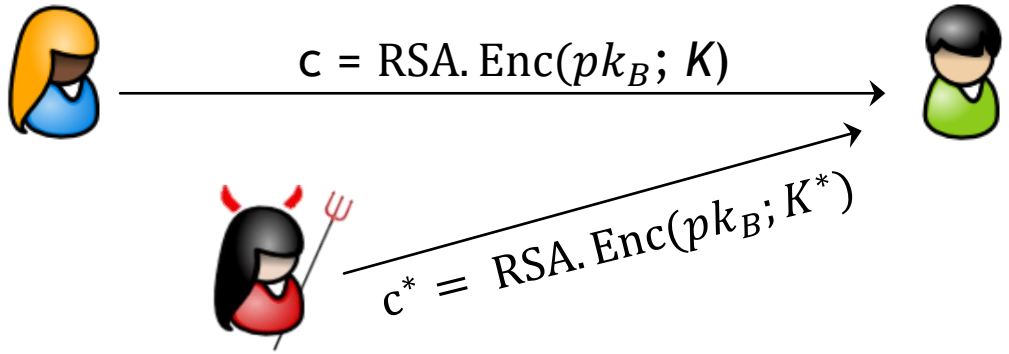


- Alice et Bob n'ont aucun indice de la presence de l'attaquant
- Alice et Bob vont maintenant utiliser les clés calculées pour chiffrer de l'information

$$m = \mathrm{Dec}(K; c)$$

Pour RSA, ce n'est pas le cas

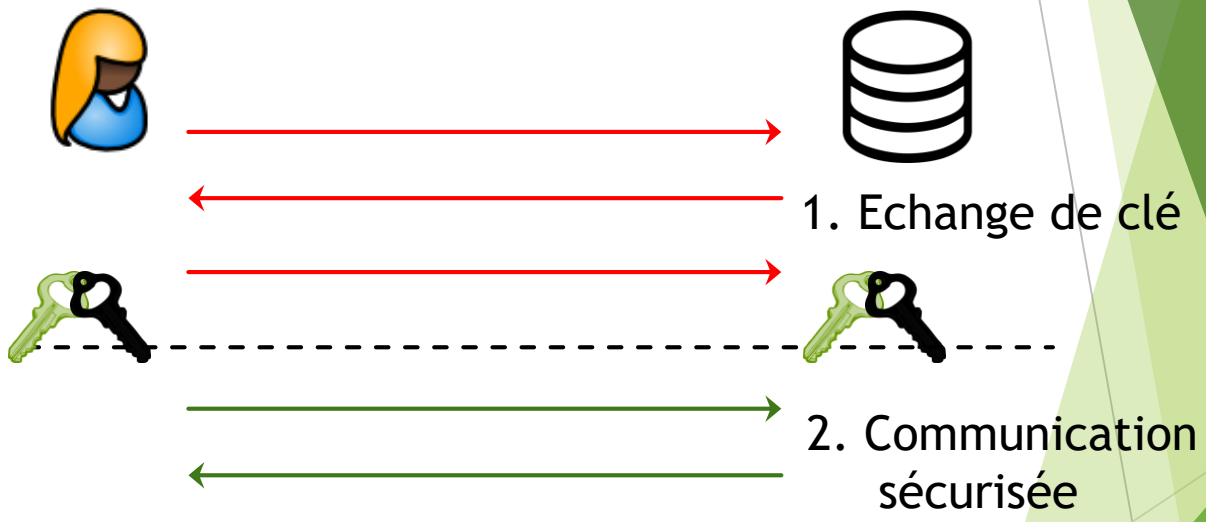
- Un attaquant peut toujours jouer le role d'un client...
  - ... mais ceci est inevitable avec une authentification unilaterale



- Si Alice chiffre maintenant des messages avec K, alors l'attaquant ne les déchiffrera pas
- Par contre, sans un autre moyen d'authentification, Bob ne saura pas avec qui il parle

Eléments principaux

- 3 éléments prinxipaux :
  - L'échange de clé authentifié
  - **La dérivation des clés**
  - L'utilisation de clés pour une communication sécurisée



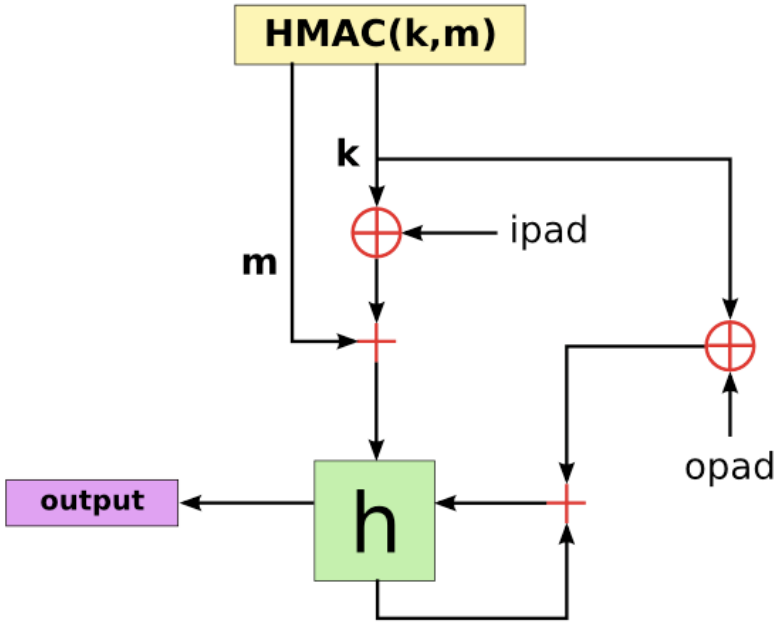
Fonction de dérivation de clé (KDF)

- Prend en **entrée** du matériel de clé & **sort** un nombre de clés indépendantes
- **Utile** pour un nombre de raisons :
  - le matériel de clé est **court** et à haute entropie, tandis que la sortie est souvent **large**, tout en maintenant une bonne entropie
  - On peut toujours générer du matériel de clé **à différents niveaux** (pour chiffrer des parties du protocole d'échange de clé, pour mettre à jour regulièrement les clés utilisées pour le chiffrement authentifié, etc.)

- **Propriété essentielle** : indistinguabilité d'une fonction aléatoire
  - Pour deux entrées "un peu différentes" les sorties sont "complètement différentes"
- Les KDFs les plus utilisées **en pratique** : HMAC [RFC 2104], HKDF [RFC 5869]

HMAC

- Les entrées de HMAC :
  - Message m
  - Clé (matériel de clé) : k
- ipad : input padding
- opad : output padding
- h : function de hachage (SHA2, SHA3)



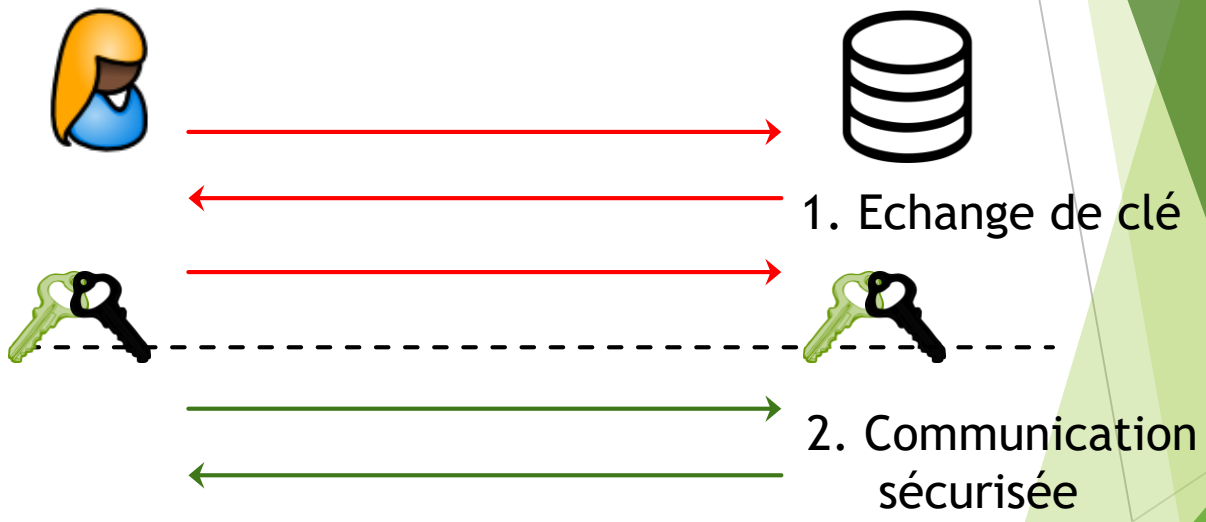
HKDF : Extract, then expand

- 2 opérations : Extract, Expand
- Philosophie :
  - On extrait du matériel de clé (haute entropie)...
  - ... puis on fait une expansion pour obtenir des clés

- HKDF utilise HMAC internellement :
  - $\mathrm{HKDF.Extract}$: utilise une valeur "sel" et une clé : $\mathrm{HMAC}(sel, clé)$
    - Soit le sel, soit la clé doit rester privé(e)
  - $\mathrm{HKDF.Expand}$: utilise une clé k (obtenue par extraction) sur un message (label)
    - La clé doit rester privée

Eléments principaux

- 3 éléments prinxipaux :
  - L'échange de clé authentifié
  - La dérivation des clés
  - **L'utilisation de clés pour une communication sécurisée**

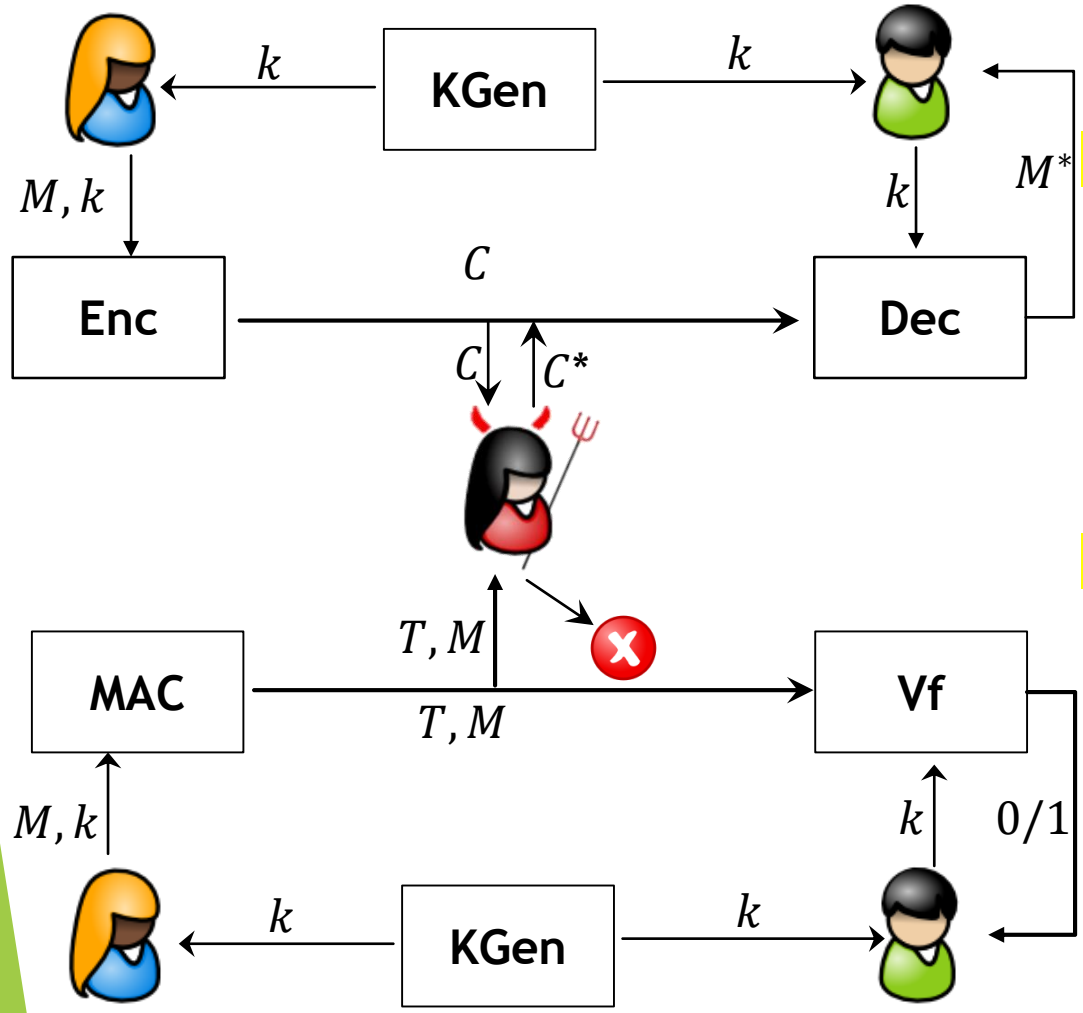


Le chiffrement authentifié

- Permet à la fois de garantir la **confidentialité** et l'**authenticité** des messages échangés sur un canal publique
  - Condidentialité : l'attaquant ne peut plus que deviner chaque bit du message
  - Authenticité : l'attaquant ne peut pas envoyer un chiffrement au nom de qq'un d'autre

- **Méthodes connues** de chiffrement authentifiés :
  - Basés sur des schéma de MAC + des schémas de chiffrement symétrique
  - AEAD : authenticated encryption with additional data

Rappel : MAC, Enc



Un schéma de chiffrement symétrique :

- Alice et Bob partagent une clé privée
- Alice chiffre son message avec la clé
- Bob déchiffre avec la même clé.

Un schéma de MAC :

- Alice et Bob partagent une clé privée
- Alice crée une tag pour un message avec sa clé
- Bob vérifie la validité du tag avec sa clé

MAC + Enc

- On considère un schéma de MAC et un schéma de chiffrement symétrique
  - Avec deux clés k_{MAC} et k_{Enc}

- Laquelle est la meilleure option ?
  - **MAC and Encrypt** d'un message M : Alice envoie :

    $(\mathrm{Enc}(k_{\mathrm{Enc}}; M) \;\; ; \;\; \mathrm{MAC}(k_{MAC}; M))$

    
  - **MAC then Encrypt** d'un message M : Alice envoie :

    $(\mathrm{Enc}\big(k_{\mathrm{Enc}}; M, \mathrm{MAC}(k_{MAC}; M)\big))$

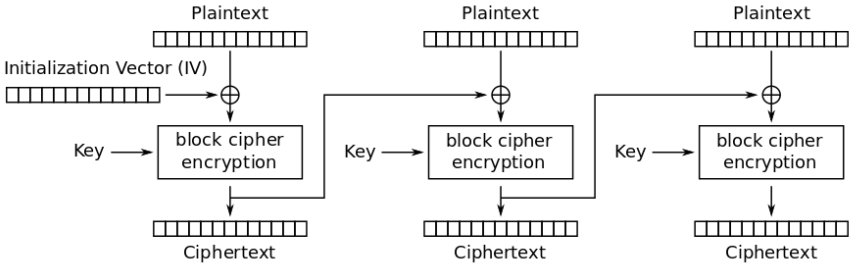
    
  - **Encrypt then MAC** d'un message M : Alice envoie :

    $(\mathrm{c} = \mathrm{Enc}(k_{\mathrm{Enc}}; M) \;\; ; \;\; \mathrm{MAC}(k_{MAC}; c))$

Rappel : les modes de chiffrement/MAC

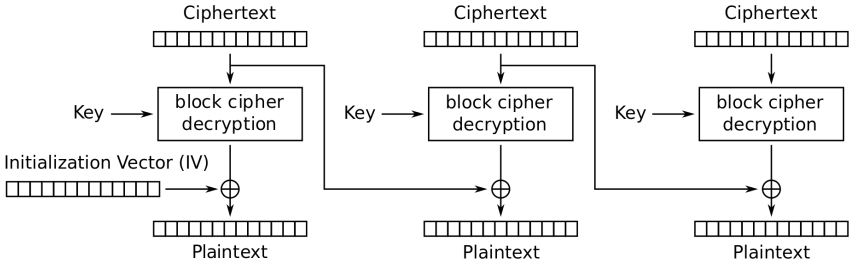
- En pratique le mode le plus utilisé en pratique, c'est le mode CBC
- Chiffrement :



Cipher Block Chaining (CBC) mode encryption

Source : Wikipedia.org

- Déchiffrement :



Cipher Block Chaining (CBC) mode decryption

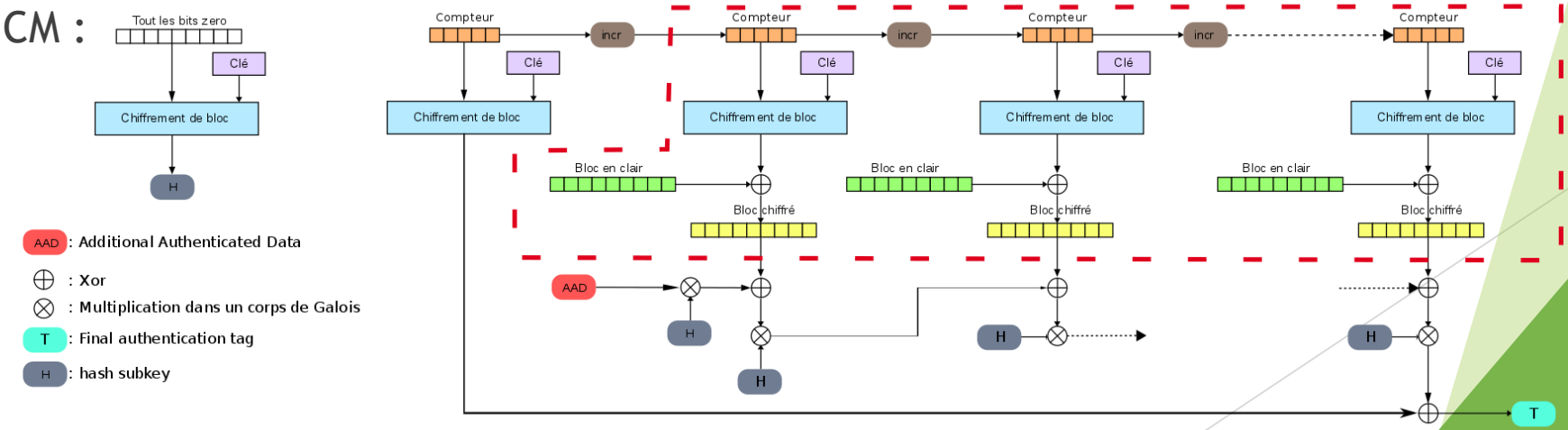
Source : Wikipedia.org

Le mode CBC est problématique pour TLS, à voir plus tard!

AEAD

- Une notion de chiffrement authentifié (AE) qui a un champ de plus (AD)
  - AE : les données resteront confidentielles ET authentifiées
  - AD : les données sont authentifiées, mais non-cofidentielles

- En TLS 1.3 : AES-GCM (Galois/Counter mode), ChaCha/Poly1305
- AES-GCM :



Le protocole TLS 1.2 -- L'échange et dérivation de clé

TLS-RSA, TLS-DH, TLS-DHE

Structure générale

- **Entrées** : le serveur a une paire de clés (sk, pk)
- **Préambule** : **é**change d'informations temporaires pour la fraîcheur de la session
  échange de paramètres de la session
- **Clés** : **o**n obtient un pre-master secret : pms
  puis, on obtient le matériel de clé, un master secret ms
  on calcule des clés pour la session à partir de ms et du préambule
- **Confirmation de clé** : le client et le serveur vérifient qu'ils ont calculé la même clé

Échange de clé TLS

Choisir $n_c \xleftarrow{\$} \{0,1\}^{224}$

n_C, opt_C →

← $n_S, \text{opt}_S, \text{KE}_S$

Choisir $n_S \xleftarrow{\$} \{0,1\}^{256}$

Vérifier $\text{opt}_S \in \text{opt}_C$

$\text{KE}_C, \{Fin_C\}_K$ →

Calculer pms
Dériver $\text{ms} = \text{HMAC}(\text{pms}; n_C, n_S)$
Dériver $\text{K} := \text{HMAC}(\text{ms}; n_c, n_S)$

Calculer pms
Dériver ms, K
Vérifier et déchiffrer $\{Fin_C\}_K$

← $\{Fin_S\}_K$

Vérifier et déchiffrer $\{Fin_S\}_K$

$\mathbf{opt_C}$	Options sur les algorithmes d'échange de clé et chiffrement authentifié
$\mathbf{opt_S}$	Le serveur choisit une option parmi celles offertes par le serveur
$\{\}_K$	Chiffrement symétrique authentifié avec la clé K

Échange de clé TLS

Choisir $n_c \xleftarrow{\$} \{0,1\}^{224}$

n_C, opt_C →

← $n_S, \text{opt}_S, \text{KE}_S$

Vérifier $\text{opt}_S \in \text{opt}_C$

Choisir $n_S \xleftarrow{\$} \{0,1\}^{256}$

$\text{KE}_C, \{Fin_C\}_K$ →

Calculer pms

Dériver $\text{ms} = \text{HMAC}(\text{pms}; n_C, n_S)$

Dériver $\text{K} := \text{HMAC}(\text{ms}; n_c, n_S)$

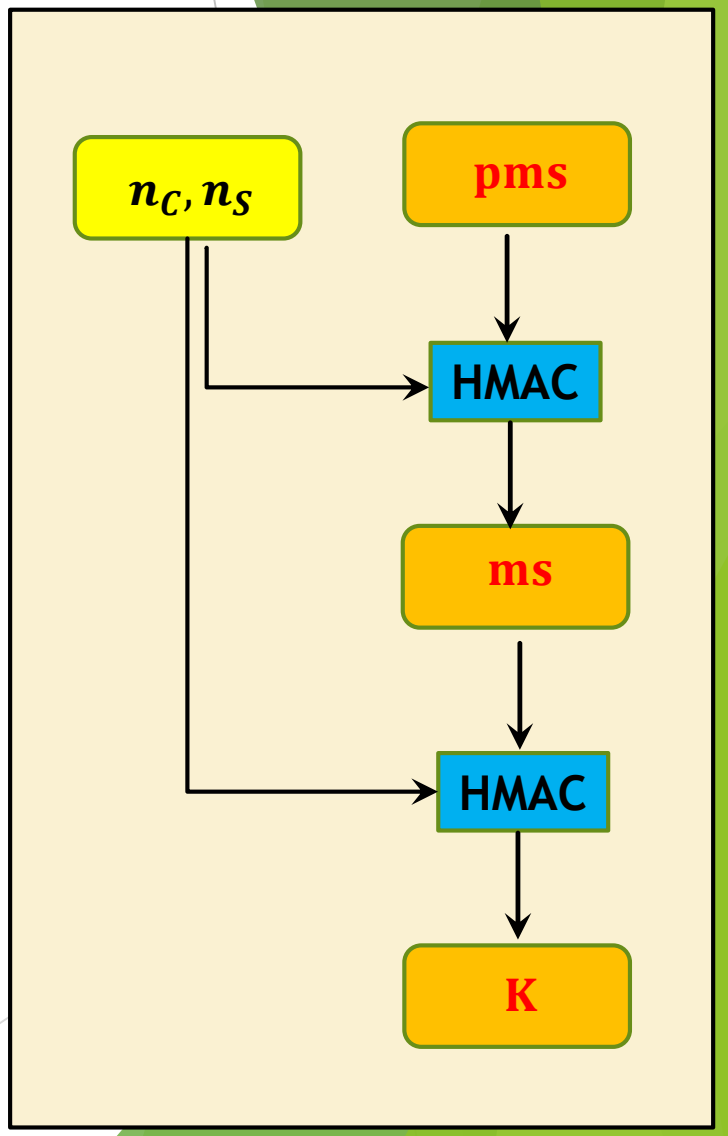
Calculer pms

Dériver ms, K

Vérifier et déchiffrer $\{Fin_C\}_K$

← $\{Fin_S\}_K$

Vérifier et déchiffrer $\{Fin_S\}_K$



TLS en mode RSA

$(sk_{\text{RSA}}, pk_{\text{RSA}})$

Choisir $n_c \xleftarrow{\$} \{0,1\}^{224}$

$n_C,\ \text{opt}_C$ →

Choisir $n_S \xleftarrow{\$} \{0,1\}^{256}$

← $n_S, \text{opt}_S, \text{KE}_S$

Vérifier Cert

Vérifier $\text{opt}_S \in \text{opt}_C$

$\text{KE}_S = (pk_{RSA}, \text{Cert})$

Choisir pms $\xleftarrow{\$} \{0,1\}^{8*48}$

$\text{KE}_C := \text{RSA.Enc}(pk_{RSA};\ \text{pms})$

$\text{KE}_C, \{\text{Fin}_C\}_K$ →

Déchiffrer KE_C à pms

Calculer ms, K, h_C, Fin_C

Vérifier $\{\text{Fin}_C\}_K$ et Fin_C

$h_S := H(n_C, \ldots KE_C, h_C)$

$\text{Fin}_S := \text{HMAC}(ms; l_2, h_S)$

Compute ms, K

$h_C := H(n_C, \ldots KE_C)$

$Fin_C := \text{HMAC}(ms; l_1, h_C)$

← $\{\text{Fin}_S\}_K$

Message	Contenu
$(sk_{\text{RSA}}, pk_{\text{RSA}})$	Paire de clés RSA certifiées
Cert	Un certificat authentifiant le serveur en tant que le propriétaire de sk_{RSA}

TLS en mode RSA

$(sk_{\text{RSA}}, pk_{\text{RSA}})$

Choisir $n_c \xleftarrow{\$} \{0,1\}^{224}$

n_C, opt_C →

Choisir $n_S \xleftarrow{\$} \{0,1\}^{256}$

← $n_S, \text{opt}_S, \text{KE}_S$

Vérifier Cert

Vérifier $\text{opt}_S \in \text{opt}_C$

$\text{KE}_S = (pk_{RSA}, \text{Cert})$

Choisir pms $\xleftarrow{\$} \{0,1\}^{8*48}$

$\text{KE}_C := \text{RSA.Enc}(pk_{RSA}; \text{pms})$

$\text{KE}_C, \{\text{Fin}_C\}_K$ →

Déchiffrer KE_C à pms

Calculer ms, K, h_C, Fin_C

Vérifier $\{\text{Fin}_C\}_K$ et Fin_C

$h_S := H(n_C, \ldots KE_C, h_C)$

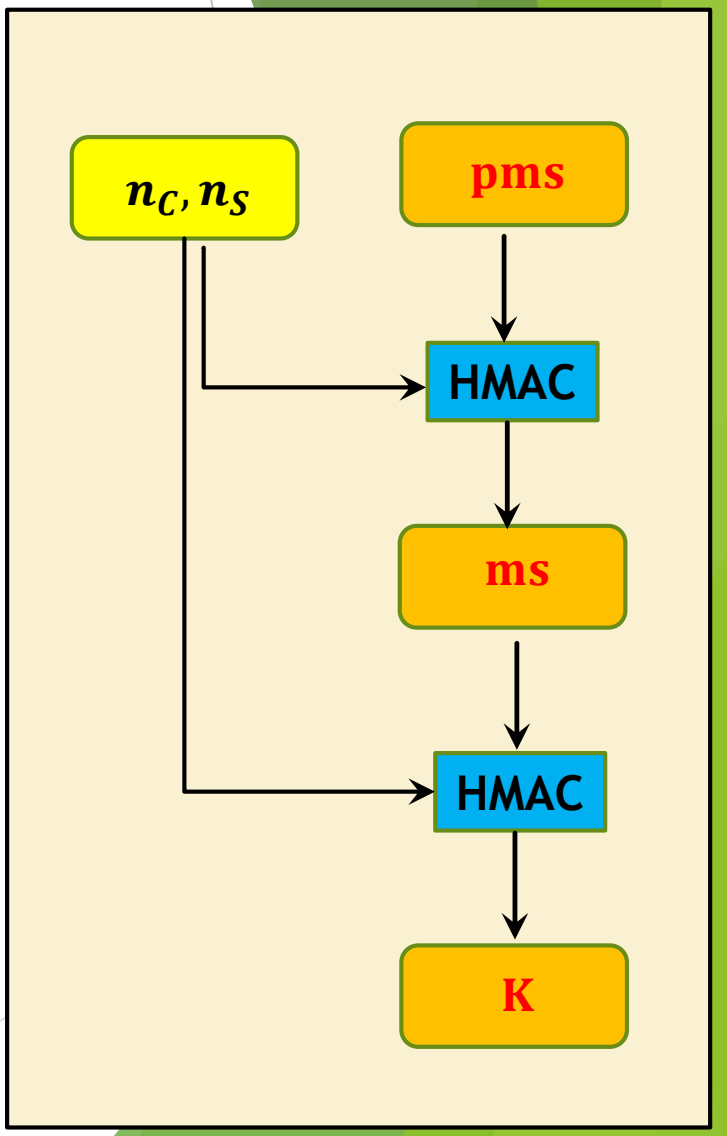
$\text{Fin}_S := \text{HMAC}(ms; l_2, h_S)$

Compute ms, K

$h_C := H(n_C, \ldots KE_C)$

$Fin_C := \text{HMAC}(ms; l_1, h_C)$

← $\{\text{Fin}_S\}_K$



TLS en mode DH

(G, p, q, s, g^s)

Choisir $n_c \xleftarrow{\$} \{0,1\}^{224}$

et $c \xleftarrow{\$} \{1, \dots, p-1\}$

n_C, opt_C →

Choisir $n_S \xleftarrow{\$} \{0,1\}^{256}$

$\text{KE}_S = (g^s, \text{Cert})$

← $n_S, \text{opt}_S, \text{KE}_S$

Vérifier Cert

Vérifier $\text{opt}_S \in \text{opt}_C$

Choisir $\text{KE}_C := g^c$, calculer $\text{pms} = (g^s)^c \bmod p$

Calculer $\text{pms} = (g^c)^s \bmod p$

$\text{KE}_C, \{\text{Fin}_C\}_K$ →

Compute ms, K

Calculer ms, K, h_C, Fin_C

Vérifier $\{\text{Fin}_C\}_K$ et Fin_C

$h_C := H(n_C, \dots KE_C)$

$h_S := H(n_C, \dots KE_C, h_C)$

$Fin_C := \text{HMAC}(ms; l_1, h_C)$

← $\{\text{Fin}_S\}_K$

$\text{Fin}_S := \text{HMAC}(ms; l_2, h_S)$

Message	Contenu
(G, p, g, g^s)	Groupe G = < g > avec générateur g, d' ordre premier p
Cert	Un certificat qui authentifie les paramètres DH du serveur

TLS en mode DH

(G, p, q, s, g^s)

Choisir $n_c \xleftarrow{\$} \{0,1\}^{224}$

et $c \xleftarrow{\$} \{1, \dots, p-1\}$

n_C, opt_C →

Choisir $n_S \xleftarrow{\$} \{0,1\}^{256}$

← $n_S, \text{opt}_S, \text{KE}_S$

Vérifier Cert

Vérifier $\text{opt}_S \in \text{opt}_C$

$\text{KE}_S = (g^s, \text{Cert})$

Choisir $\text{KE}_C := g^c$, calculer $\text{pms} = (g^s)^c \bmod p$

Calculer $\text{pms} = (g^c)^s \bmod p$

$\text{KE}_C, \{\text{Fin}_C\}_K$ →

Compute ms, K

Calculer ms, K, h_C, Fin_C

Vérifier $\{\text{Fin}_C\}_K$ et Fin_C

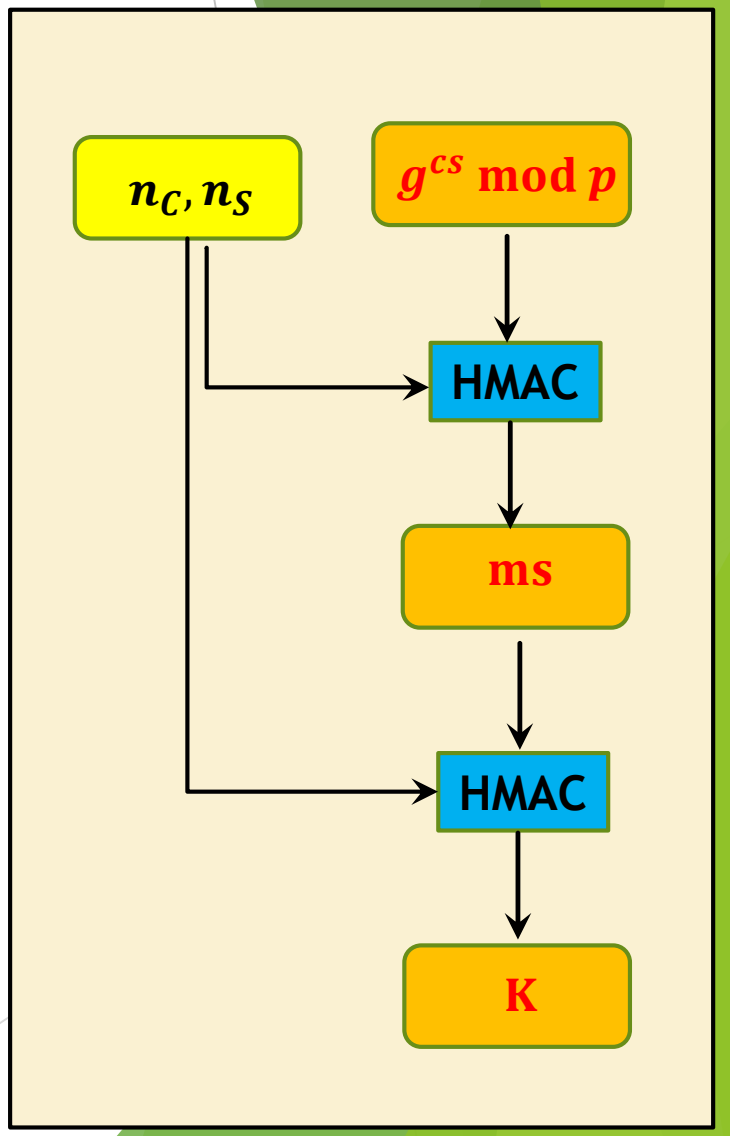
$h_C := H(n_C, \dots KE_C)$

$h_S := H(n_C, \dots KE_C, h_C)$

$\text{Fin}_C := \text{HMAC}(ms; l_1, h_C)$

← $\{\text{Fin}_S\}_K$

$\text{Fin}_S := \text{HMAC}(ms; l_2, h_S)$



TLS en mode DHE

(sk_{Sign}, pk_{Sign})

Choisir $n_c \xleftarrow{\$} \{0,1\}^{224}$

et $c \xleftarrow{\$} \{1, \dots, p-1\}$

n_C, opt_C →

Choisir $n_S \xleftarrow{\$} \{0,1\}^{256}$

Choisir paramètres DH :
p, $G = < g >$, g^s
Calculer
$p = H(n_C, n_S, p, g, g^s)$,
$\sigma = Sign(g^s, p)$
$KE_S = (p, g, g^s, \sigma, Cert)$

← $n_S, \text{opt}_S, \text{KE}_S$

Vérifier Cert
Vérifier $\text{opt}_S \in \text{opt}_C$

Choisir $KE_C := g^c$, calculer
$pms = (g^s)^c \bmod p$

$\text{KE}_C, \{\text{Fin}_C\}_K$ →

Calculer $pms = (g^c)^s \bmod p$

Compute ms, K

$h_C := H(n_C, \dots KE_C)$

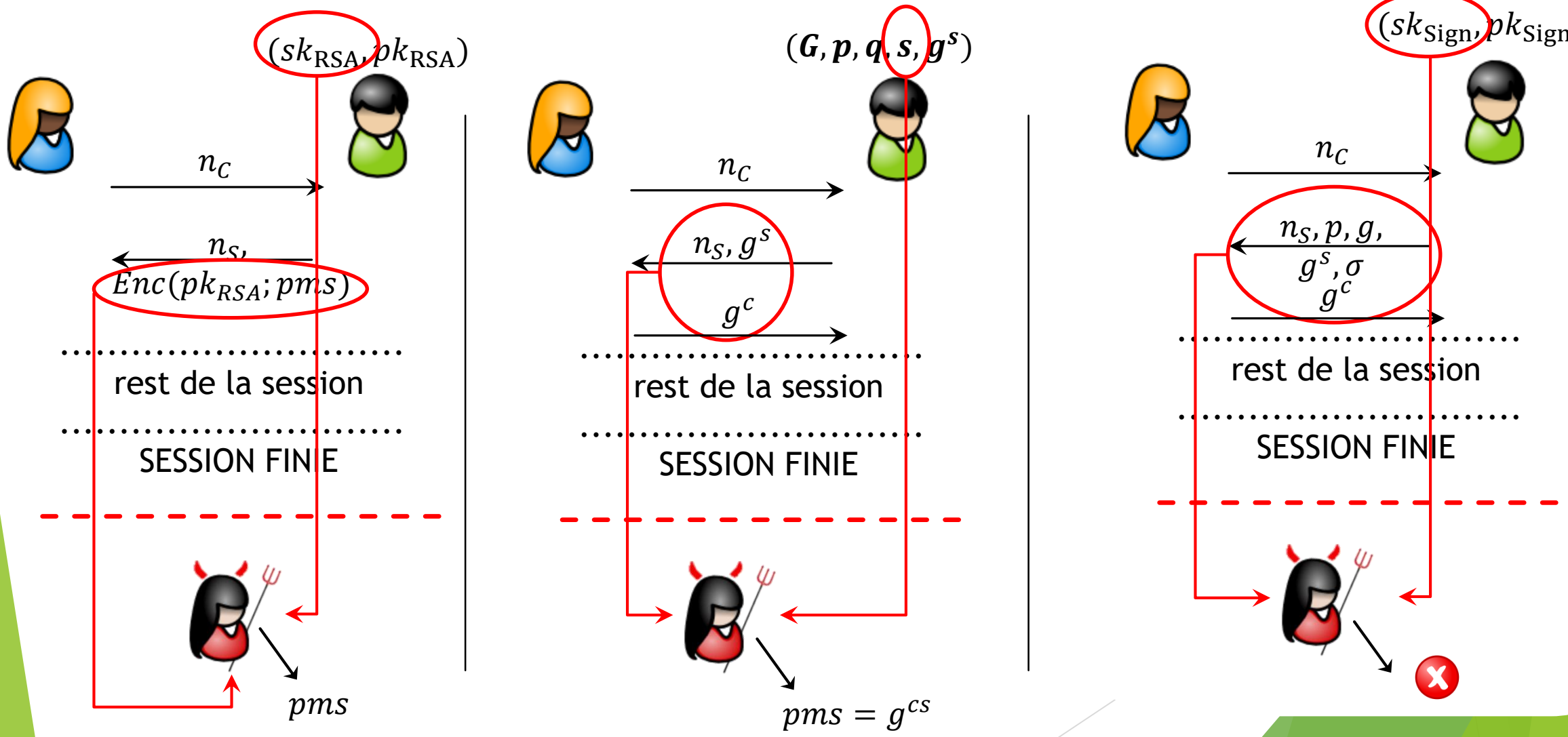
$Fin_C := \text{HMAC}(ms; l_1, h_C)$

← $\{\text{Fin}_S\}_K$

Calculer ms, K, h_C, Fin_C
Vérifier $\{\text{Fin}_C\}_K$ et Fin_C
$h_S := H(n_C, \dots KE_C, h_C)$
$\text{Fin}_S := \text{HMAC}(ms; l_2, h_S)$

Message	Contenu
(sk_{Sign}, pk_{Sign})	Paire de clés pour un schéma de signatures
Cert	Un certificat qui authentifie les clés de signature

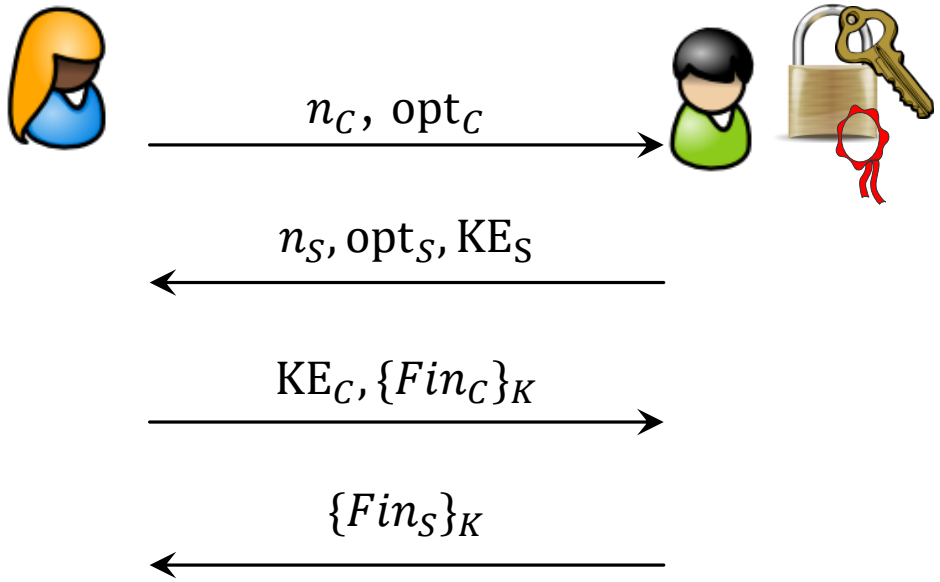
PFS pour les trois modes



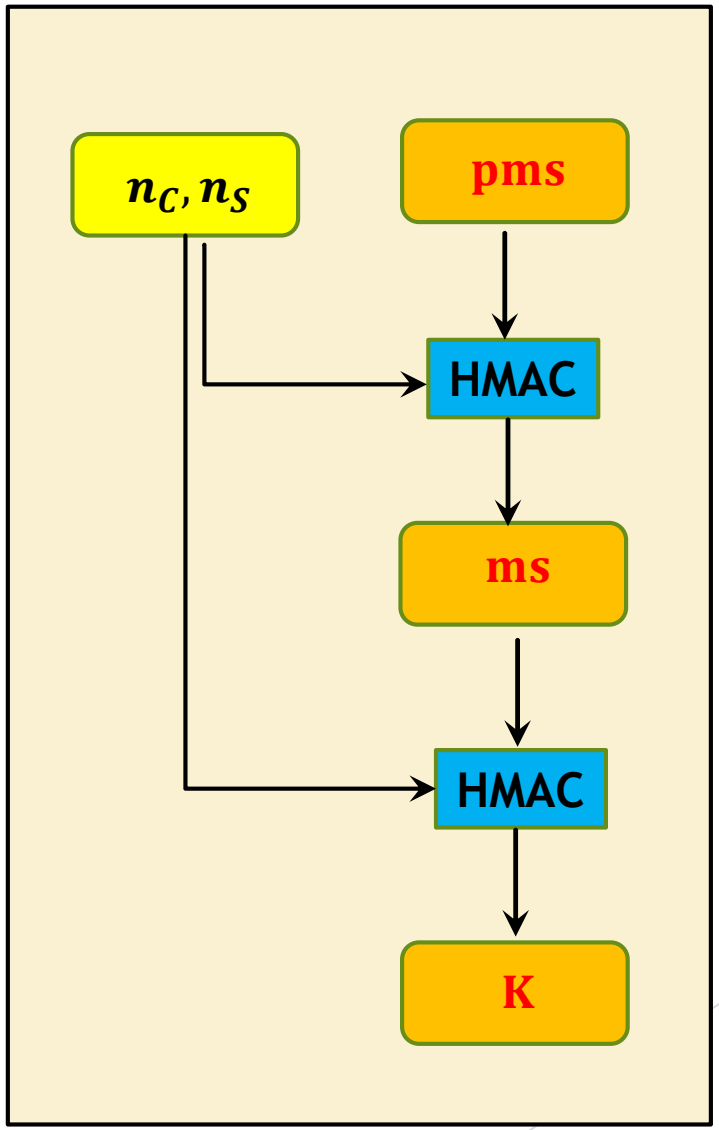
TLS Session Resumption

- Disons qu'un client et un serveur ont établi une connexion
  - Cette connexion est ensuite fermée
- Lorsqu'ils parlent à nouveau ils ont deux options :
  - Un nouvel établissement d'un canal sécurisé
  - Un établissement abrévié du canal sécurisé dit "Session resumption"
- Concept : on utilise le matériel de clé d'une session passée pour calculer des nouvelles clés avec une nouvelle fraîcheur

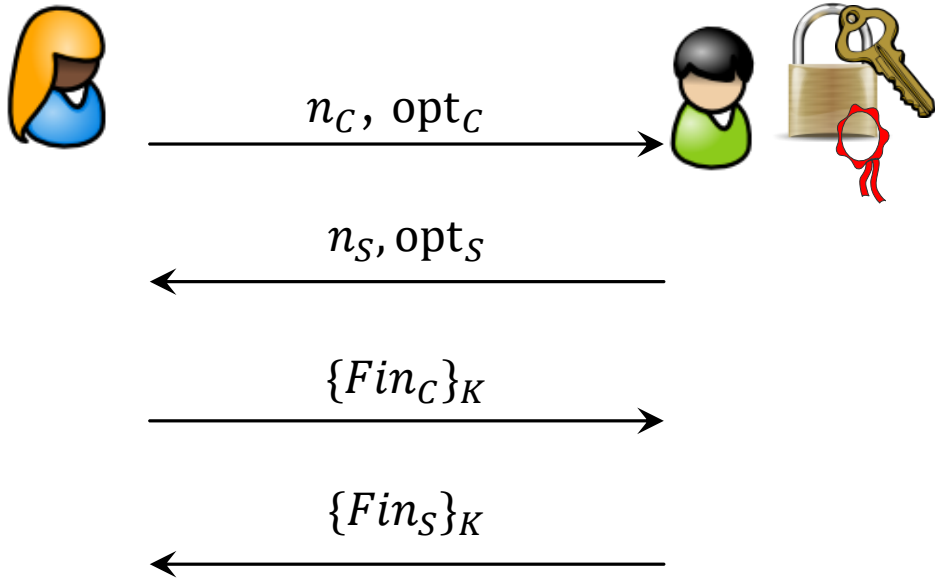
Session resumption



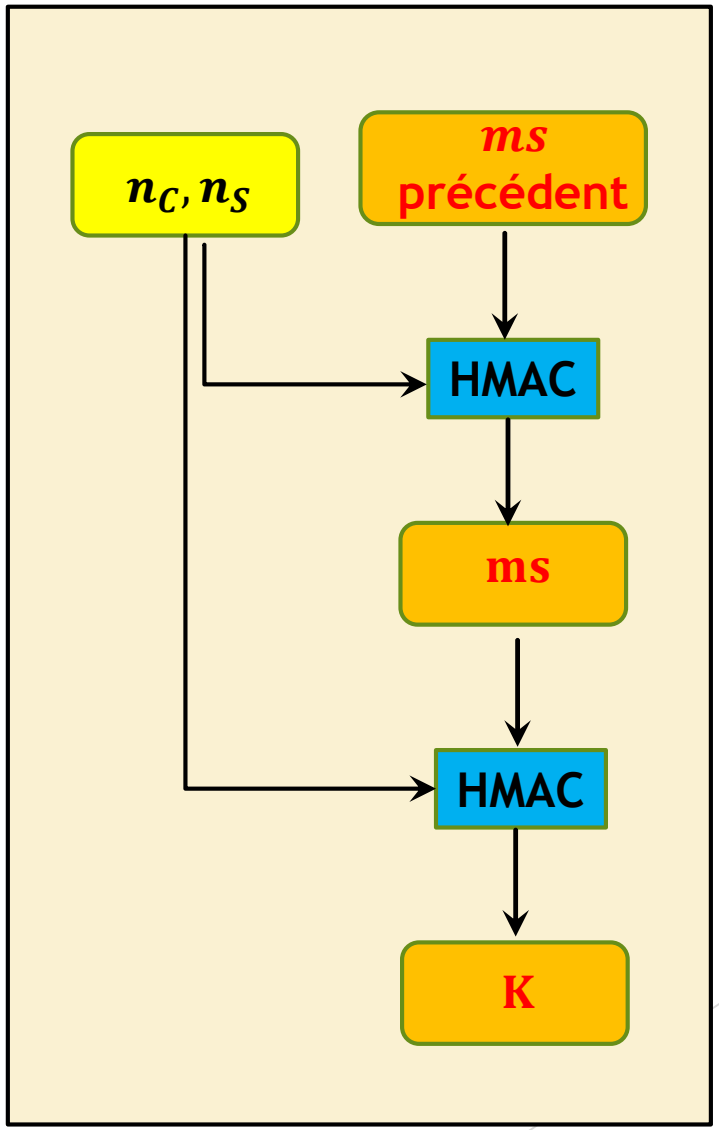
Session complète



Session abbrégée



Session abbregée



Les options et les records TLS

Versions, Cipher suites, padding,

Les versions de TLS

- Le protocole que vous avez vu est la version TLS 1.2, la plus utilisée aujourd'hui
- Mais TLS n'a pas commencé à cette version...

  - SSL 1.0: jamais publiée (trop peu sécurisée...)
  - SSL 2.0: publiée Fev. 1995, avec "un nombre de failles de sécurité"
  - SSL 3.0: publiée en 1996, nouveau design

  - TLS 1.0: “plus sécurisée, pas de grosses modifs”
  - TLS 1.1: un peu plus de sécurité contre les algorithmes faibles
  - TLS 1.2: publiée en 2008, l'année où on a cassé MD5
    MD5 n'a été éradiquée de TLS que vers 2014...

"Backward compatibility"

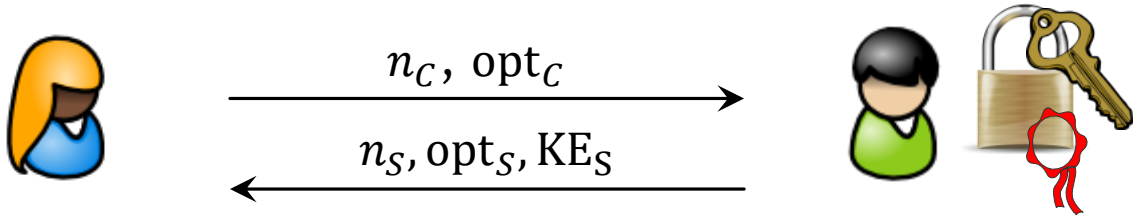
- Les versions de TLS ont progressé, à chaque fois étant plus sécurisées
  - En fait, les nouvelles versions réparaient souvent des failles existantes
- Pourtant, à chaque fois, les anciennes versions restaient en usage
  - Backward compatibility: les clients et serveurs peu mis à jour doivent pouvoir communiquer, donc les anciennes versions sont encore implémentées
- Il y a 5 ans, on pouvait encore se connecter à un site web en utilisant SSL 3.0
  - Ce qui est étonnant étant donné le nombre de failles qu'il présente

Est-ce que cela n'affecte qu'un nombre limité de serveurs, trop peu mis à jour ?

Version rollback

- Une attaque qui "force" un client et un serveur de négocier une version vulnerable

Qu'est-ce qui se passe lorsque l'option du serveur ne match pas les options du client?



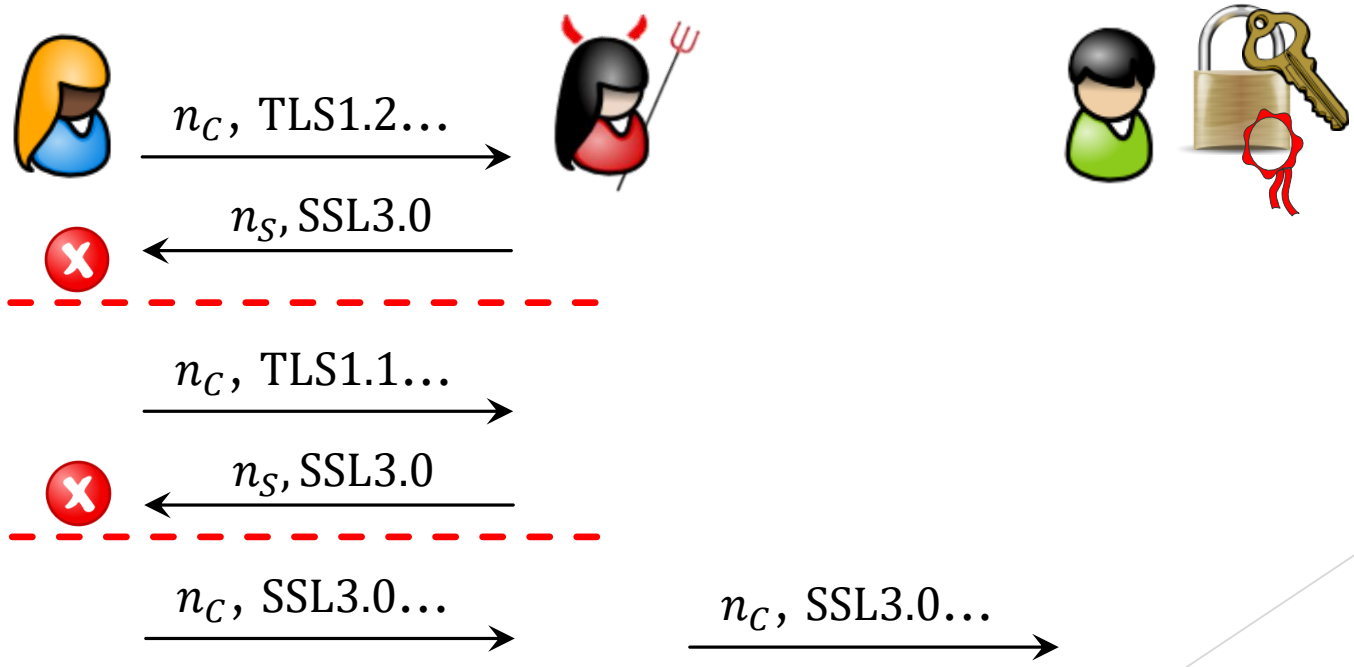
Vérifier $\mathbf{opt}_S \in \mathbf{opt}_C$

Sinon, on recommence!

Version rollback

- Une attaque qui "force" un client et un serveur de négocier une version vulnerable

Qu'est-ce qui se passe lorsque l'option du serveur ne match pas les options du client?



Les options TLS 1.2

- Le client envoie une liste d'options d'algorithmes (cipher suite)

CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA256 = { 0x00,0x3C };

CipherSuite TLS_DH_RSA_WITH_AES_128_CBC_SHA = {0x00, 0x31};

CipherSuite TLS_DHE_DSS_WITH_AES_256_CBC_SHA = { 0x00,0x38 };

- Une cipher suite contient un nombre d'éléments
  - Partie avant le mot clé WITH : échange de clé
  - Partie après le mot clé WITH : l'algorithme de chiffrement authentifié, avec la fonction de hachage utilisée

Les options TLS 1.2

- Le client envoie une liste d'options d'algorithmes (cipher suite)

CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA256 = { 0x00,0x3C };

CipherSuite TLS_DH_RSA_WITH_AES_128_CBC_SHA = {0x00, 0x31};

CipherSuite TLS_DHE_DSS_WITH_AES_256_CBC_SHA = { 0x00,0x38 };

- Partie échange de clé :
  - Mode RSA : les deux mots TLS_RSA
  - Modes DH, DHE : trois mots, TLS_DH ou TLS_DHE avec un algorithme de signature qui certifie les éléments

Les options TLS 1.2

- Le client envoie une liste d'options d'algorithmes (cipher suite)

CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA256 = { 0x00,0x3C };

CipherSuite TLS_DH_RSA_WITH_AES_128_CBC_SHA = {0x00, 0x31};

CipherSuite TLS_DHE_DSS_WITH_AES_256_CBC_SHA = { 0x00,0x38 };

- Partie chiffrement authentifié :
  - Algorithme de chiffrement
  - Taille de la clé
  - Mode de chiffrement
  - Fonction de hachage

Négotiation des paramètres

- Le client fait une liste de versions et cipher suites dans l'ordre de sa preference
- Le serveur reçoit ce message
  - Soit il n'accepte aucune des options possibles (et le protocole peut recommencer)
  - Soit il finit par en accepter une

- On connait deux types de serveurs :
  - Les serveurs **directifs** : choisit son option préférée parmi celles offertes par le client, même s'il pourrait en accepter une plus haute dans la liste du client
  - Les serveurs **courtois** : choisit la première option possible parmi celles offertes par le client, même si celle-ci n'est pas l'option préférée du serveur

Quelle(s) pourrai(en)t être les avantages et inconvénients de ces comportements ?

L'authentification des paramètres

Choisir $n_c \xleftarrow{\$} \{0,1\}^{224}$

n_C, opt_C →

← $n_S, \text{opt}_S, \text{KE}_S$

Vérifier $\text{opt}_S \in \text{opt}_C$

Choisir $n_S \xleftarrow{\$} \{0,1\}^{256}$

$\text{KE}_C, \{Fin_C\}_K$ →

Calculer pms

$\text{ms} = \text{HMAC}(\text{pms}; n_C, n_S)$

$\text{K} := \text{HMAC}(\text{ms}; n_c, n_S)$

Calculer pms

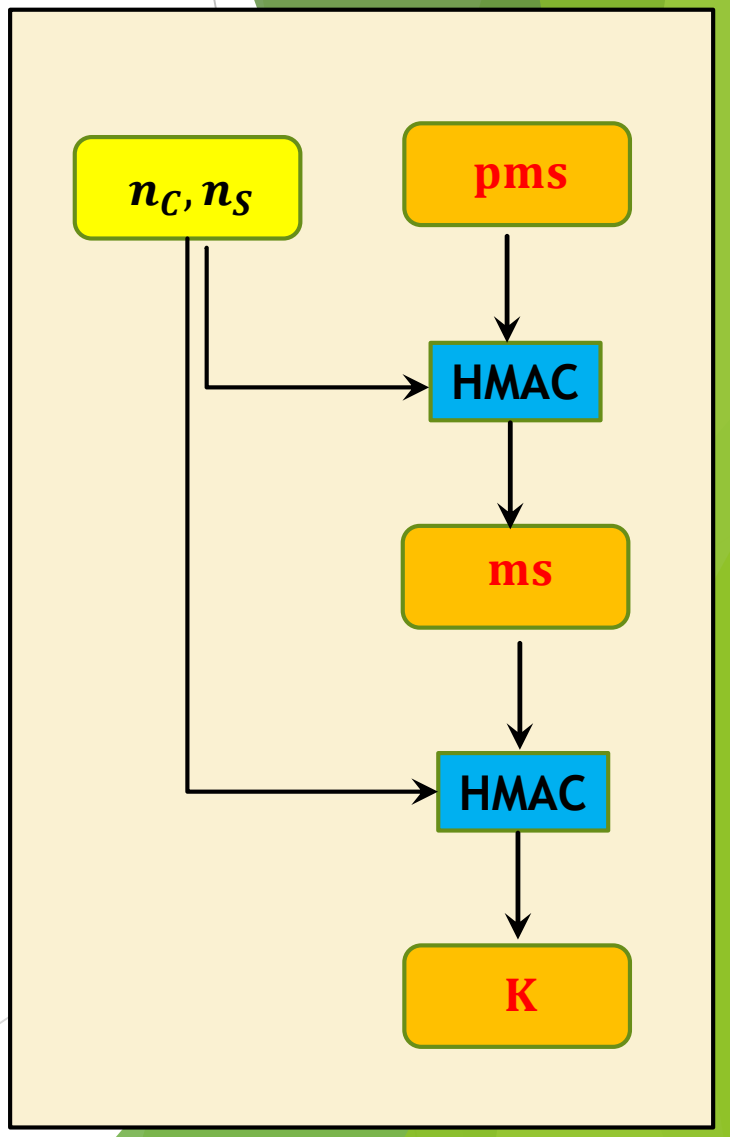
Dériver ms, K

Vérifier et déchiffrer $\{Fin_C\}_K$

← $\{Fin_S\}_K$

Vérifier et déchiffrer $\{Fin_C\}_K$

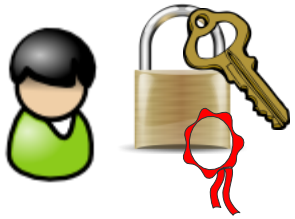
Les options ne sont pas mises en entrée des clés !



Les attaques par downgrade

- Les modes RSA, DH, DHE ont existé depuis du début de TLS
  - Mais pas avec les mêmes tailles de paramètres
- Initialement TLS faisait partie des logiciels sensibles, que les Etats Unis ne voulait pas qu'on utilise à la même capacité ailleurs
  - D'où les cipher suites dites d'export -- paramètres de taille réduite
- Les attaques par downgrade se basent sur deux choses seulement:
  - Le fait que ces modes export persistent toujours dans l'implém
  - Le fait que les options ne sont pas utilisés pour le calcul de la clé

L'attaque FREAK



$(sk_{\mathrm{RSA}}, pk_{\mathrm{RSA}})$

n_C, $[RSA \ldots]$

n_C

$[RSA.exp \ldots]$

Choisir $n_c \xleftarrow{\$} \{0,1\}^{224}$

$n_S, [RSA], \mathrm{KE}_S$

$n_S, [RSA.exp], \mathrm{KE}_S$

Choisir $n_S \xleftarrow{\$} \{0,1\}^{256}$

$\mathbf{KE_S = (pk_{RSA}, Cert)}$

Vérifier Cert

Vérifier $\mathrm{opt}_S \in \mathrm{opt}_C$

Choisir pms $\xleftarrow{\$} \{0,1\}^{8*48}$

Factorise le module

La clé RSA n'aura que 512 bits

Compute *ms*, *K*

$\mathrm{KE}_C, \{\mathrm{Fin}_C\}_K$

$\mathrm{KE}_C, \{\mathrm{Fin}_C^*\}_K$

Trouve pms

Déchiffrer KE_C à pms

Calculer $ms, K, h_C, \mathrm{Fin}_C$

$\{\mathrm{Fin}_S\}_K$

$\{\mathrm{Fin}_S^*\}_K$

L'attaque LogJam

n_C, [DHE …]

n_C,
[DHE, Exp …]

(G, p, q, s, g^s)

Choisir $n_c \xleftarrow{\$} \{0,1\}^{224}$
et $c \xleftarrow{\$} \{1, \dots, p-1\}$

Vérifier Cert
Vérifier $\text{opt}_S \in \text{opt}_C$

n_S, DHE,
KE_S

n_S, DHE.Exp,
KE_S

Choisir $n_S \xleftarrow{\$} \{0,1\}^{256}$

Paramètres DH 512 bits :
p, $G = < \text{g} >$, g^s

Trouve le dlog s

Choisir $\text{KE}_C := g^c$, calculer
$\text{pms} = (g^s)^c \bmod p$

Calculer
$p = H(n_C, n_S, p, g, g^s)$,
$\sigma = Sign(g^s, \text{p})$
$\text{KE}_S = (p, g, g^s, \sigma, \text{Cert})$

Compute ms, K

$\text{KE}_C, \{\text{Fin}_C\}_K$

$\text{KE}_C, \{\text{Fin}_C^*\}_K$

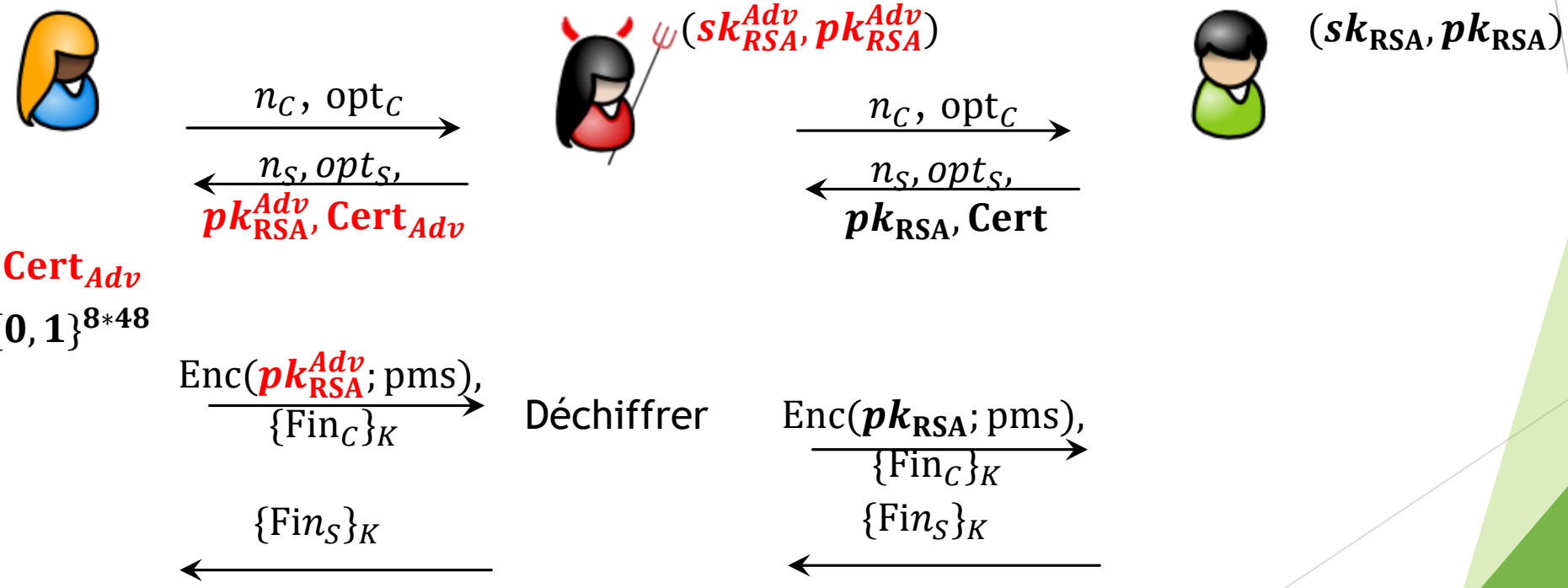
Calculer ms, K,

$\{\text{Fin}_S^*\}_K$

$\{\text{Fin}_S\}_K$

Avec un serveur malveillant 3SHAKE

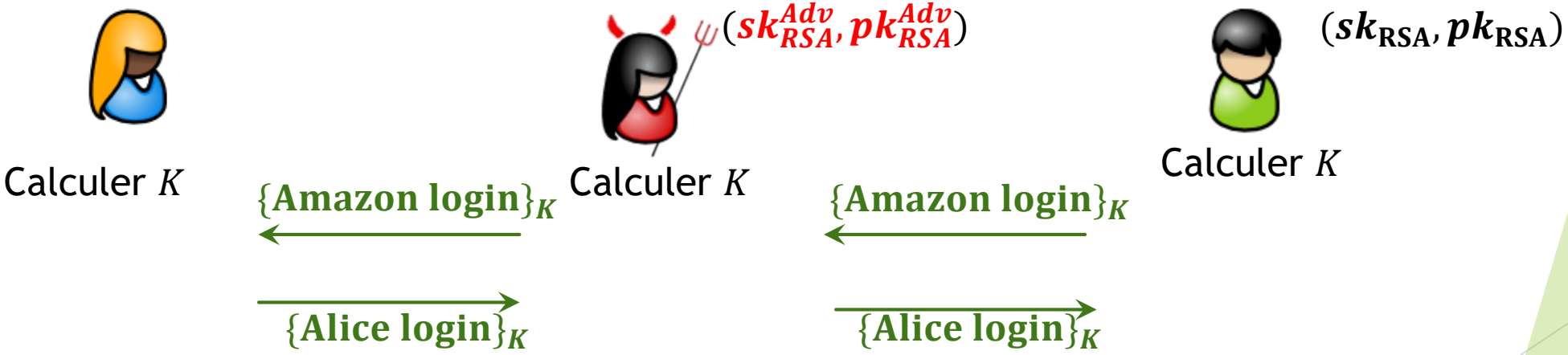
- Imaginons l'administrateur d'un domaine Web https://monDomaine.fr
- https = http + TLS, donc l'administrateur possède une paire de clés légitimes



Faisable (plus difficile) pour DH(E)

3SHAKE

- Imaginons l'administrateur d'un domaine Web https://monDomaine.fr
- https = http + TLS, donc l'administrateur possède une paire de clés légitimes



Des attaques sur la couche "record"

Le mode CBC, RC4, CRIME/BREACH, etc.

A partir de maintenant...

On suppose que le client et le serveur ont déjà effectué l'échange de clé

Supposons même que la clé est bonne

Ceci ne veut pas dire que la communication sera sécurisée...

Le bourrage CBC en TLS

- Dans le mode CBC les messages à chiffrer doivent être bourrés pour avoir une taille qui est un multiple de la taille d'un bloc AES
- Le bourrage a un format précis :
  - Si on doit ajouter un seul octet,celui-ci sera 0x01
  - Si on doit ajouter deux octets, ceux-ci seront 0x02
  - Etc.

- Dans le chiffrement CBC, si on déchiffre et le bourrage est incorrect, le message d'erreur spécifie cela

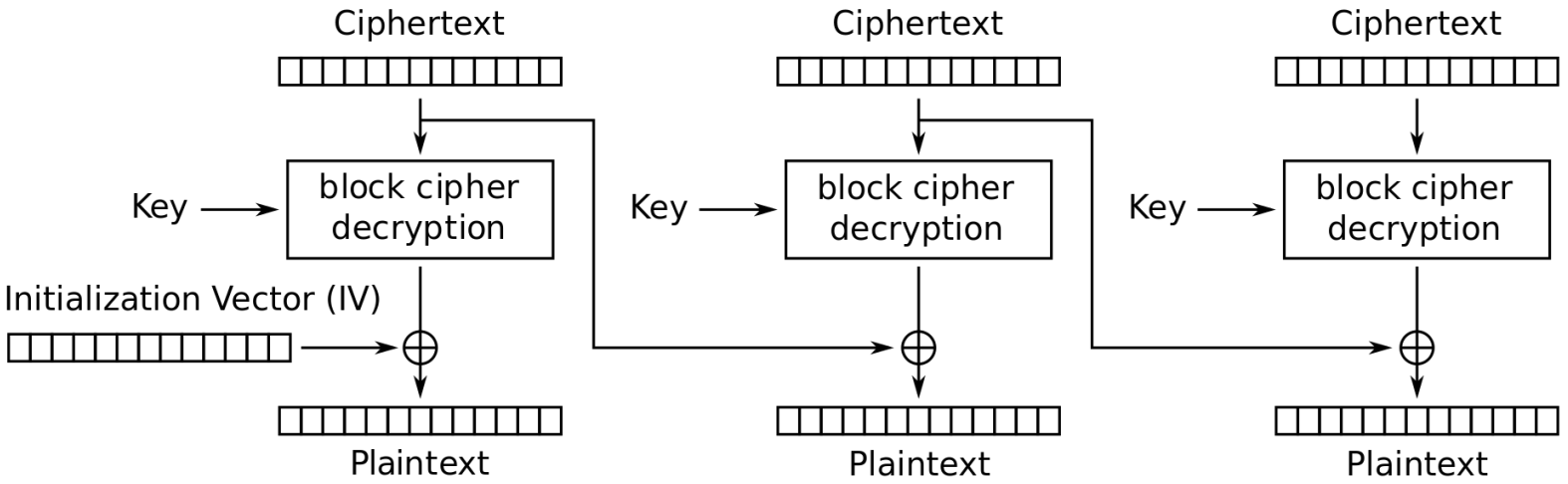
Un oracle de bourrage

- Imaginons une machine initialisée avec une clé de chiffrement (qu'on ne connait pas) et une valeur de bourrage souhaitée
- A chaque fois qu'on lui donne un chiffré (AES-CBC) elle nous renvoie un seul bit (0 si le bourrage est faux, 1 s'il est bon)
- Cette machine s'appelle L'oracle de Bourrage de Vaudenay
  - En réalité le role de cette machine peut être joué par un serveur qui exécute TLS et qui a déjà calculé une clé de session

Le bourrage et le mode CBC

- Un bloc AES a 16 octets
- Le déchiffrement du dernier bloc se fait à partir de deux derniers blocs :

$$\boldsymbol{P_n = C_{n-1} + \mathbf{Dec}(K; C_n)}$$



Cipher Block Chaining (CBC) mode decryption

Source: wikipedia.org

Le bourrage et le mode CBC

- Un bloc AES a 16 octets
- Le déchiffrement du dernier bloc se fait à partir de deux derniers blocs :

$$\boldsymbol{P_n = C_{n-1} \oplus \mathbf{Dec}(K; C_n)}$$

- Nous commençons avec un chiffré valide (envoyé, disons, par le vrai client)
- Nous allons manipuler C_{n-1} pour fixer le dernier bit de P_n à un bourrage de 0x01
  - Nous allons utiliser l'oracle de Vaudenay initialisé avec la vraie clé K et le bourrage 0x01

Stratégie de l'attaque

- Le déchiffrement du dernier bloc se fait à partir de deux derniers blocs :

$$\boldsymbol{P_n = C_{n-1} \oplus \mathbf{Dec}(K; C_n)}$$

- Nous prenons le (vrai) dernier octet de C_{n-1} et nous le modifions à une autre valeur
- On teste le chiffré avec l'oracle de Vaudenay
- Tant que l'oracle ne retourne pas 1 (bourrage valide), on continue à modifier le dernier octet de C_{n-1}

- Disons que finalement on a un C^*_{n-1} tel que :

$$********** \boldsymbol{0x01 = C^*_{n-1} \oplus \mathbf{Dec}(K; C_n)}$$

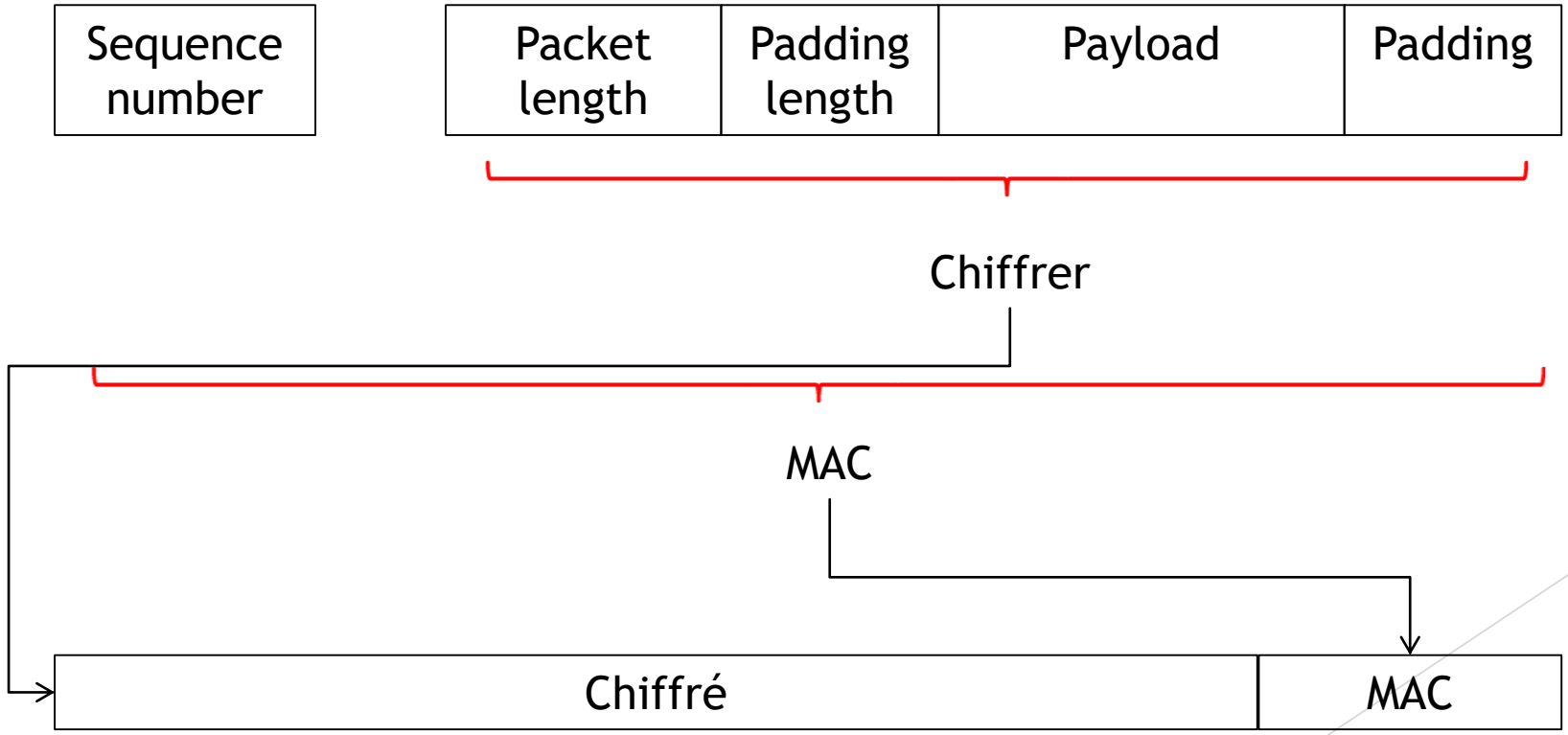
Ceci nous donne le dernier octet de $\mathbf{Dec}(K; C_n)$

Continuer l'attaque

- **On continue pour l'avant dernier, avec un bourrage de 0x02 0x02**
  - On connait actuellement le dernier octet de $\mathbf{Dec}(\boldsymbol{K}; \boldsymbol{C_n})$
  - Donc on connait la valeur du dernier octet de C_{n-1} qui nous donnerait 0x02
  - On réitère l'attaque en changeant l'avant dernier octet du bloc déjà modifié de C_{n-1}
- Puis on continue, octet par octet...
- Attaque efficace car au max 256 possibilités à chaque fois
- Utilité directe : connaître des petits morceaux d'un message, e.g. cookies (BEAST)

Et si le bourrage est non-regulier

- Albrecht et al. a montré une attaque similaire sur Encrypt-and-MAC (dans SSH, mais généralement applicable en mode CBC)



L'attaque sur Encrypt-then-MAC

- Albrecht et al. a montré une attaque similaire sur Encrypt-and-MAC

- Disons qu'on a n'importe quel texte chiffré C^* avec son MAC M^*
- Le chiffrement du mode CBC respecte l'ordre des parties du message
  - Donc le chiffré de la partie "taille du message" sera dans le premier bloc
- On prend le premier bloc de C^*
- Nous allons essayer trouver le plaintext

Astuce de l'attaque

- Pour Encrypt-and-Mac le destinataire reçoit (C^*, M^*) packet par packet
- Le destinataire ne sait pas où commence le MAC

- Il déchiffre le premier bloc pour savoir la taille du message + padding
  - Après d'avoir reçu ce que le destinataire pense sont le chiffré + le MAC...
  - ... le destinataire vérifie le MAC
  - Si celui-ci est incorrect, le destinataire envoie un message d'erreur

- On utilise le message d'erreur MAC comme oracle de déchiffrement

La structure de l'attaque

- Disons qu'on a un chiffré $C_1 C_2 \dots C_n$ qu'on veut déchiffrer
- On prend le premier block C_1 et on veut le déchiffrer

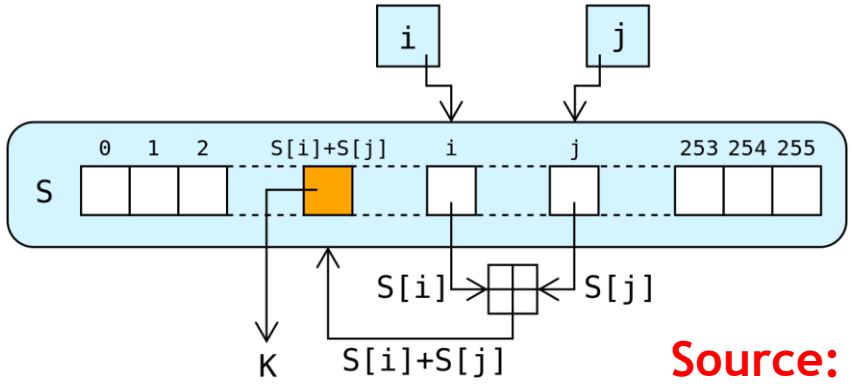
- On envoie au destinataire premièrement C_1
- Puis on commence envoyer, bloc par bloc, des messages aléatoires
- Lorsqu'on reçoit un message Erreur MAC, on s'arrête
- La taille (#blocs envoyés) nous donne une approximation du contenu de C_1

- On prend C_2 et on répète le procès

Une solution à CBC ?

- La réponse à ces attaques a été de décourager l'utilisation du mode CBC
- ... et d'encourager l'utilisation de RC4

- RC4 est un chiffre par flot :
  - On génère avec RC4 des bits qui ont l'aire aléatoire
  - Au fur et à mesure on fait un XOR bit par bit entre le message et les bits aléatoires



Source: wikipedia.org

Le problème avec RC4

- Pour un chiffrement sécurisé, les bits générés doivent sembler aléatoires
- Ceci veut dire que chaque octet à une proba d'autour de $1/256$ d'être généré par l'algorithme

- Des biais montrés par AlFardan et al. :
  - 2014: si j'ai 2^{36} chiffrés et un nombre de sessions, alors je peux te trouver les bits aléatoires et donc déchiffrer du traffic
  - 2015: avec moins de chiffrés je peux casser des mots de passe courts (6 charactères)

Les attaques ne s'arrêtent ici...

- Des attaques qui exploitent la compression de données : CRIME/BREACH
- Des attaques qui exploitent le downgrade + le bourrage : POODLE
- Des attaques qui exploitent une mauvaise implementation : Heartbleed
- Des attaques qui exploitent le bourrage avec des canaux auxiliaires : Lucky13
- ...

La solution ? TLS 1.3

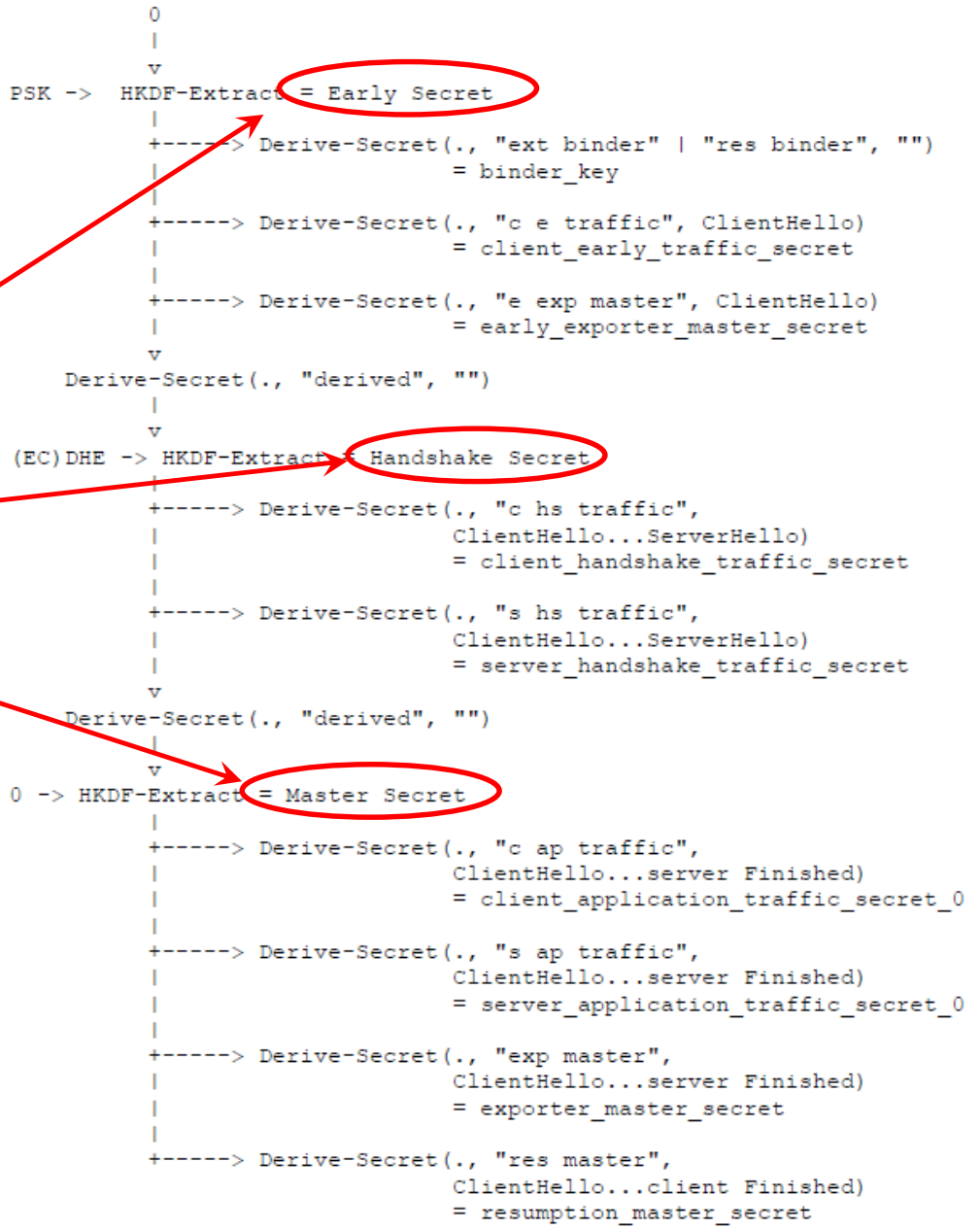
TLS 1.3 status quo

- Version déjà standardisée : RFC 8446
- Implémenté et présent en Firefox et Chrome, ainsi que dans d'autres navigateurs (mais parfois non par défaut)

- Philosophie : security-by-design, privacy-by-design
  - Se prémunir contre des failles déjà connues
    - Plus de RC4, plus de CBC
    - Les versions et options dans le calcul de la clé
    - Seulement le mode TLS-DHE
  - Plus de protection de la vie privée -- chiffrement de l'echange de clé
  - Plus d'options de session resumption : PSK, PSK+DHE, 0RTT

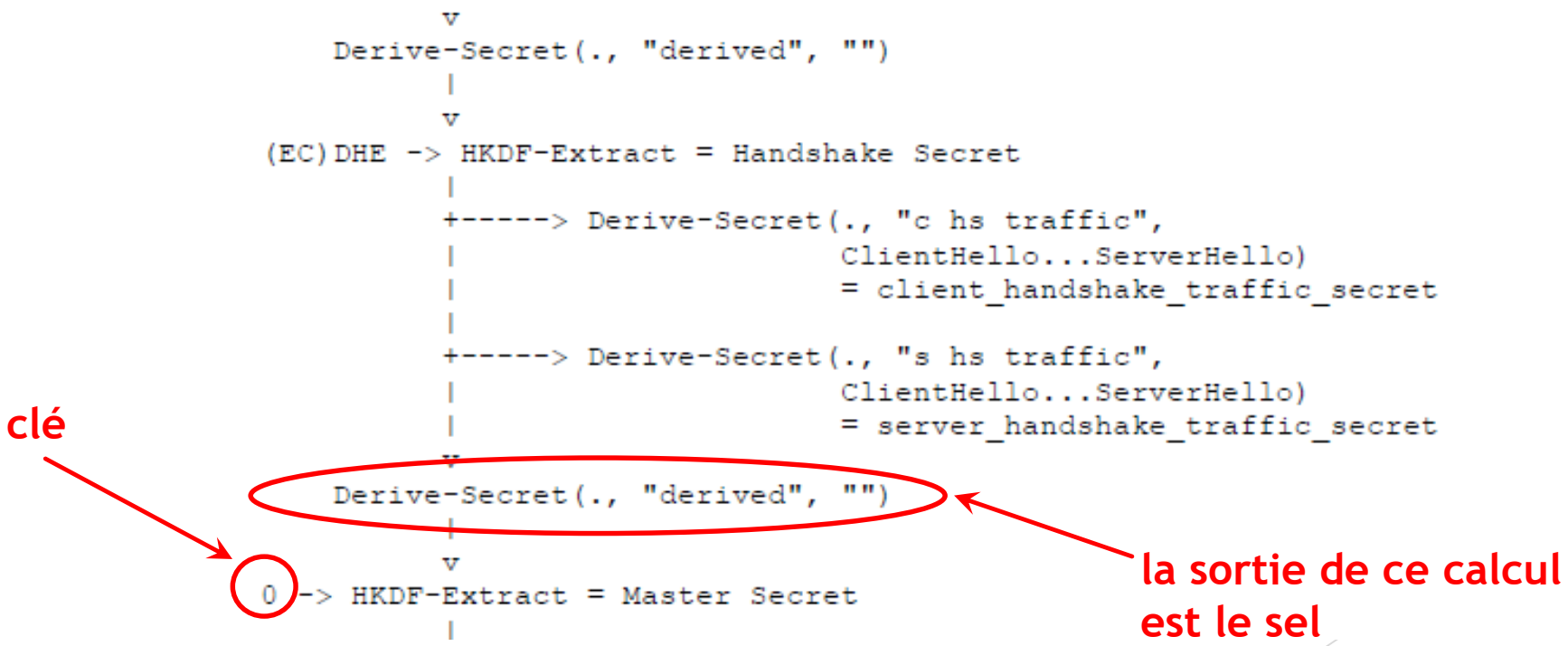
Des clés pour toute opération

- Trois secrets principaux :
  - Early secret (pour session resumption seulement)
  - Handshake secret (pour le chiffrement pendant le protocole d'échange de clé)
  - Master secret (pour la derivation des clés qui sécurisent la communication après le déroulement du protocole)
- Les secrets sont extraits avec HKDF.Extract
- A partir des secrets on dérivent des clés avec HKDF.Expand
- Chaque clé aura une entrée (dite label) différente, ainsi qu'un Hash des messages antérieures



L'utilisation de HKDF.Extract

- HKDF.Extract utilise un "sel" et une "clé"
  - Si un de ces éléments est inconnu, alors on peut assurer la sécurité des clés dérivées



L'utilisation de HKDF.Expand

- Utilisée par l'algorithme Derive-Secret
- En entrée : un secret, une "étiquette" (label) et une valeur hachée
- Propriété de HKDF.Expand :
  - Même secret, même hachée, 1 bit de différence en etiquette : deux clés différentes

```
            v
(EC)DHE -> HKDF-Extract = Handshake Secret
            |
            +-----> Derive-Secret(., "c hs traffic",
            |                     ClientHello...ServerHello)
            |                     = client_handshake_traffic_secret
            |
            +-----> Derive-Secret(., "s hs traffic",
            |                     ClientHello...ServerHello)
            |                     = server_handshake_traffic_secret
            v
      Derive-Secret(., "derived", "")
            |
```

Seulement DHE

- Le mode d'échange de clé est toujours le mode DHE
- Seulement quelques **groupes valides**
  - Ces groupes sont bien choisis et sécurisés
- Le client choisit un **sous-ensemble de groupes valides**
  - Pour chaque groupe, le client choisit **un élément** pour l'échange de clé : $c, g^c \pmod p$
- Le serveur choisit un groupe valide parmi les choix du client
  - Il répond avec un élément g^s et il signe une haché des messages envoyés, avec un préfixe
  - Le préfixe sert à assurer qu'une version rollback n'est pas possible (autre format que les autres versions)
  - Le serveur aura une paire de clés de signatures.

Échange de clé TLS

Choisir $n_c \xleftarrow{\$} \{0,1\}^{224}$

$$n_C, \text{opt}_C, [g_1^c, g_2^c \dots g_n^c] \longrightarrow$$

$$\longleftarrow n_S, \text{opt}_S, g^s$$

Vérifier $\text{opt}_S \in \text{opt}_C$

Choisir $n_S \xleftarrow{\$} \{0,1\}^{256}$

$$\longleftarrow \{\text{Cert}, \text{Sign}(sk; H[n_C, \dots g^s])\}_{S.htk}$$

$$\longleftarrow \{Fin_S\}_{S.htk}$$

Vérifier et déchiffrer $\{Fin_S\}_{S.htk}$

$$\{Fin_C\}_{C.htk} \longrightarrow$$

Vérifier et déchiffrer $\{Fin_C\}_K$

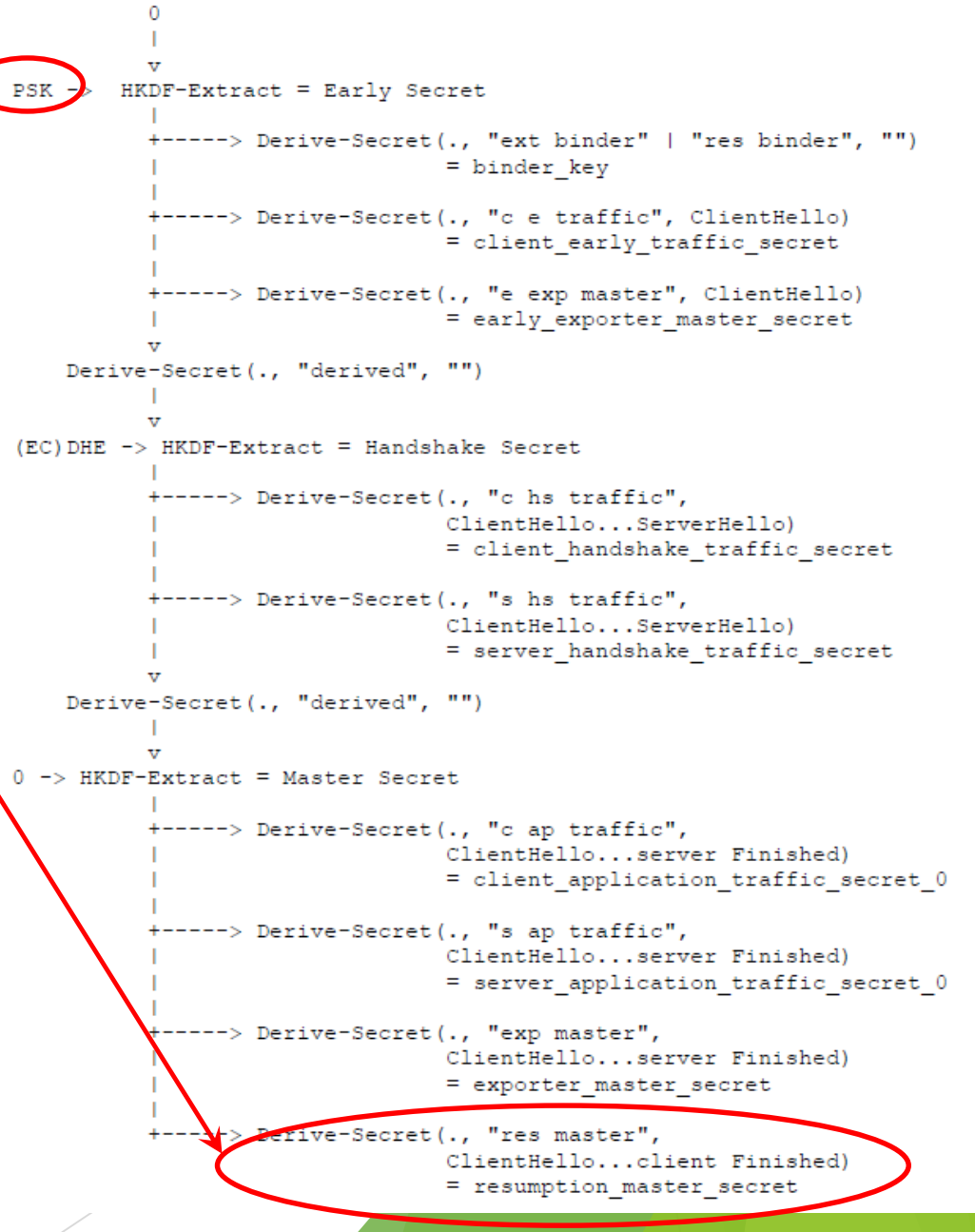
TLS 1.3 : Session Resumption

- 3 modes de session resumption : PSK, PSK+DHE, 0-RTT
- Pour chaque mode de resumption, les secrets se basent sur une valeur PSK
  - Pre-shared key
  - Calculée à partir de Resumption Master Secret (RMS) de la session passée

- Pour le **mode PSK** :
  - Le client et le serveur utilisent des nonces, mais pas de nouveaux paramètres DHE
- Pour **PSK + DHE** :
  - Le client et le serveur utilisent des nonces et de nouveaux éléments DHE
- **0-RTT** :
  - Le client et serveur utilisent les clés à partir de Early Secret

La dérivation de clé

- Dans un premier temps :
  - Le client et le serveur exécutent une session normale
  - Au bout du protocole, ils calculent le resumption master secret, à partir duquel ils pourront calculer le PSK
  - Ils indiquent s'ils veulent utiliser le rms dans le future
  - Puis la session finit
- Dans un deuxième temps :
  - Le client indique qu'il veut reprendre la session passée
  - Si le serveur est d'accord, les deux utilisent le PSK dérivé à partir du rms



Quelques propriétés de l'échange de clé

TLS 1.2	TLS 1.3
• trois modes : RSA, DH, DHE	• un mode seulement : DHE
• en mode DHE, choix libre de groupe	• groups valides établies
• le serveur choisit l'élément d'échange de clé en premier	• le client choisit l'élément d'échange de clé en premier
• 2 rondes de communication	• 1.5 rondes seulement
• session resumption à partir de ms	• session resumption en 3 modes à partir de rms
• pas de chiffrement	• chiffrement d'une partie de l'échange de clé
• options d'échange de clé non-incluses dans les clés	• options incluses dans le calcul des clés
• chiffrement des messages Finished et records avec même clé	• séparation de clés par chaque role et chaque partenaire

Chiffrement authentifié

- Plus de RC4, plus de mode CBC -- qu'est-ce qu'on utilise alors ?
- Deux options seulement :
  - AES-GCM (avec des clés plus ou moins longues) : le mode Galois Counter Mode, qui utilise un counter dans le chiffrement de chaque bloc
  - ChaCha avec Poly1305 : chiffre par flot
- Conclusion de l'équipe de design : plus d'options veulent dire plus de failles, plutôt que plus de versatilité
- Albrecht et al. ont prouvé que ces deux modes ont une bonne sécurité