TLS徹底演習

Ver. 1.0

Security Camp 2016

IIJ 大津 繁樹

2016年8月10日

自己紹介

- 株式会社インターネットイニシアティブ(IIJ)
- 経営企画本部 配信事業推進部
- オープンソースプロジェクト Node.js の Core Technical Committee メンバー、TLS/crypto 関連機能の技術担当。

本講義の目的

- TLSを徹底的に理解してもらう。

- でもTLSは各種セキュリティ技術の集合体、それぞれが深くて難しい。８時間あっても全部は無理。

- そこで３つに分けました。

    1. 座学：技術者にとってなぜこれからTLSが重要か

    2. 講義・演習： TLSハンドシェイクを学ぶ

    3. 講義・演習：TLS技術のコア、暗号技術を学ぶ

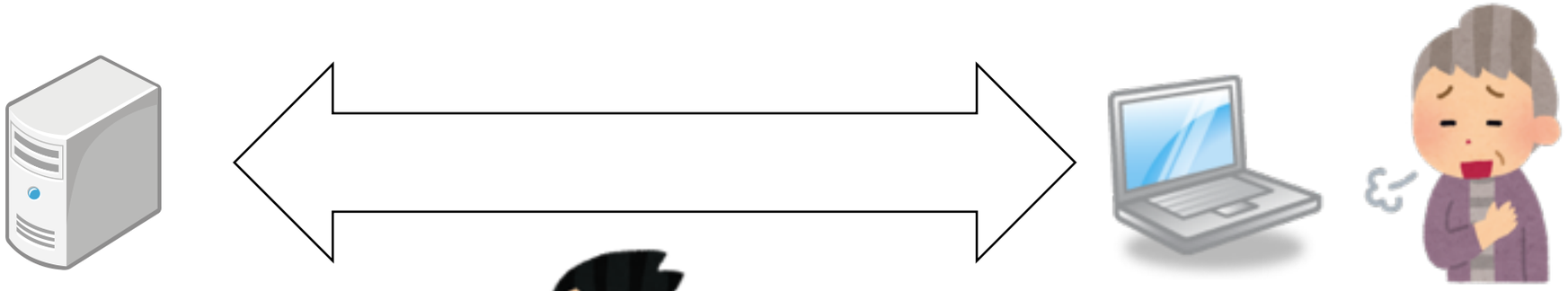
本日の講義の流れ

- 講義：TLSの概要
- 講義：TLSを理解する準備(特にAEAD)
- 講義・演習：TLSハンドシェイク説明、TLS BotとTLSハンドシェイクしよう、リアルMan-in-The-Middle
- 講義・演習: ChaCha20-Poly1305の実装

TLSの概要

インターネットの脅威

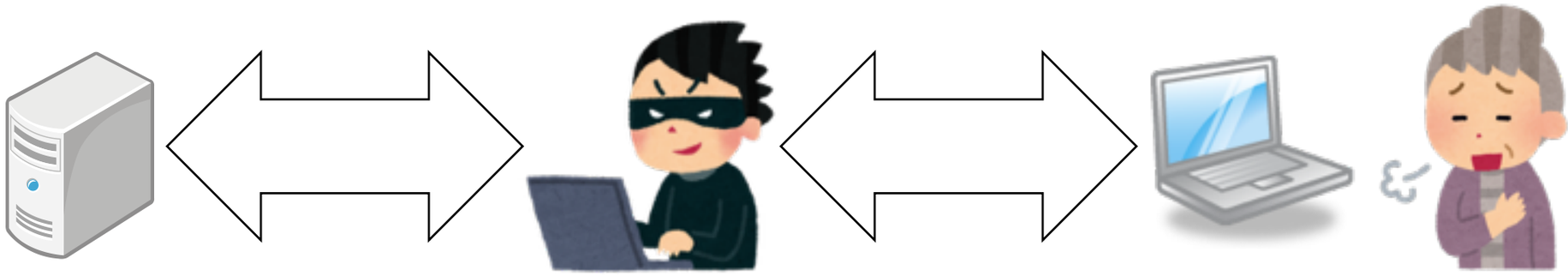
盗聴



パスワードやクレジットカード番号を盗み見

インターネットの脅威

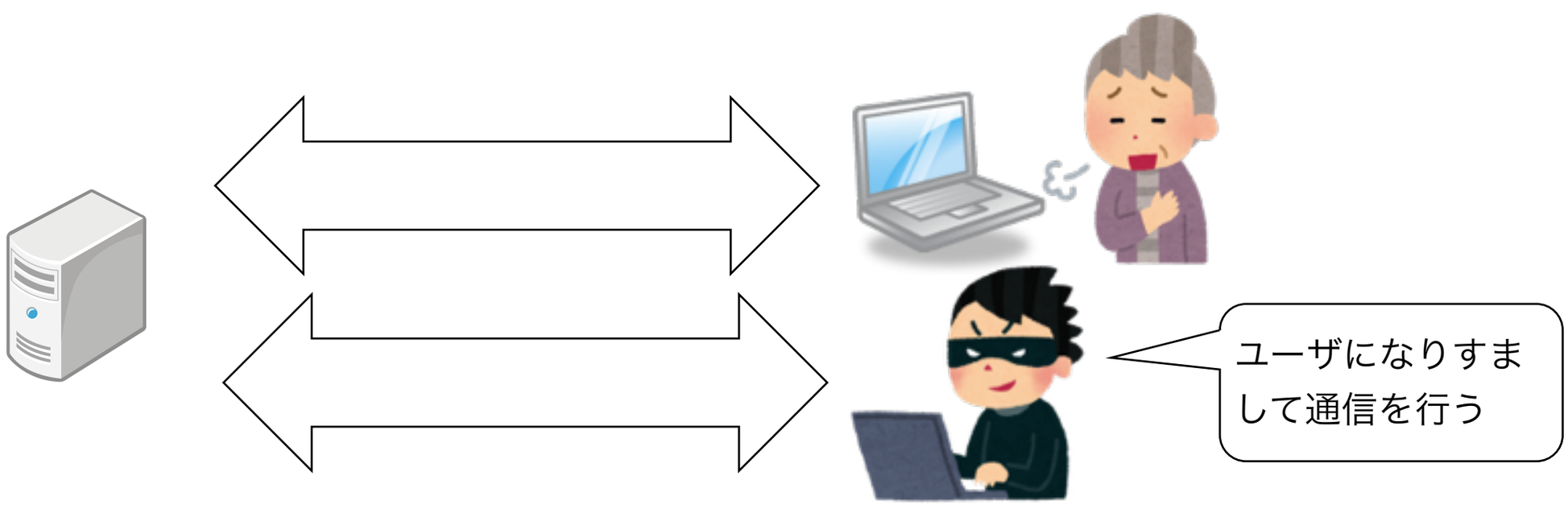
改ざん



通信途中でデータを書き換え

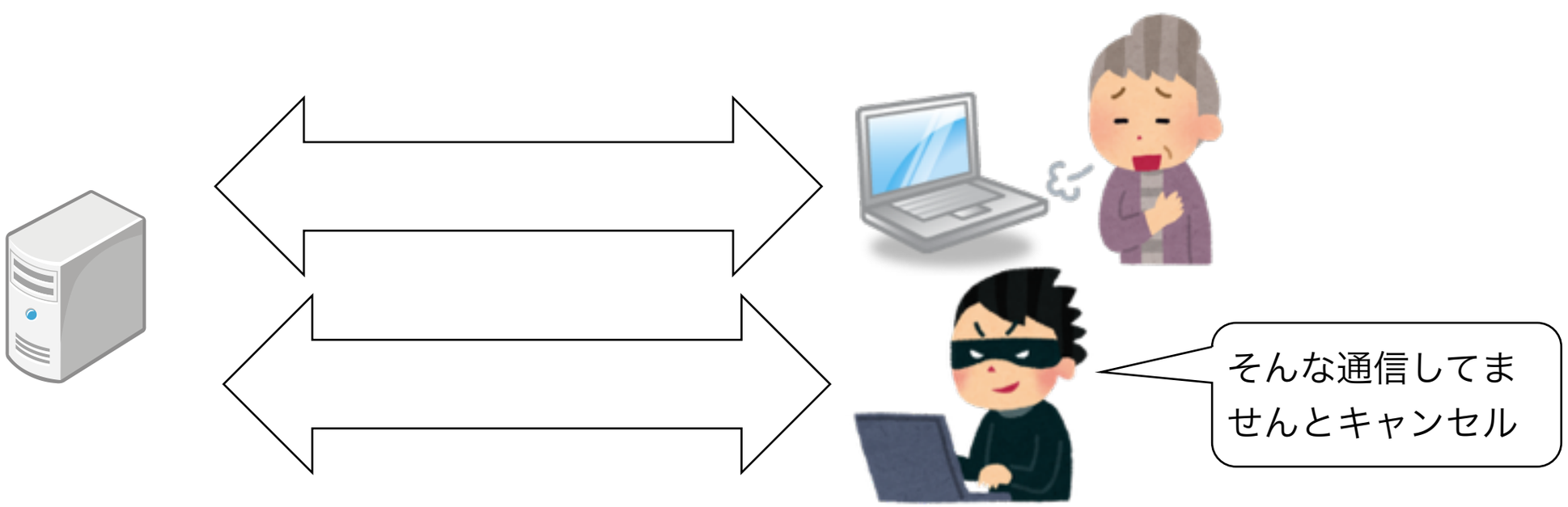
インターネットの脅威

なりすまし

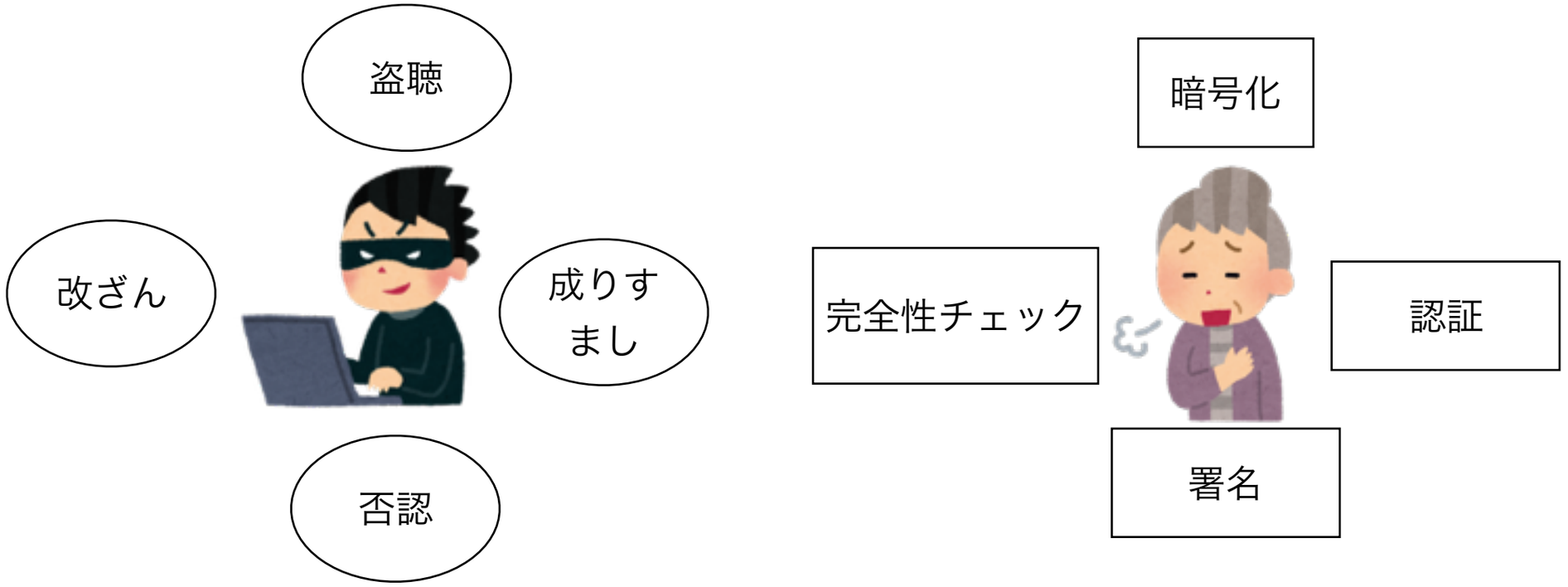


インターネットの脅威

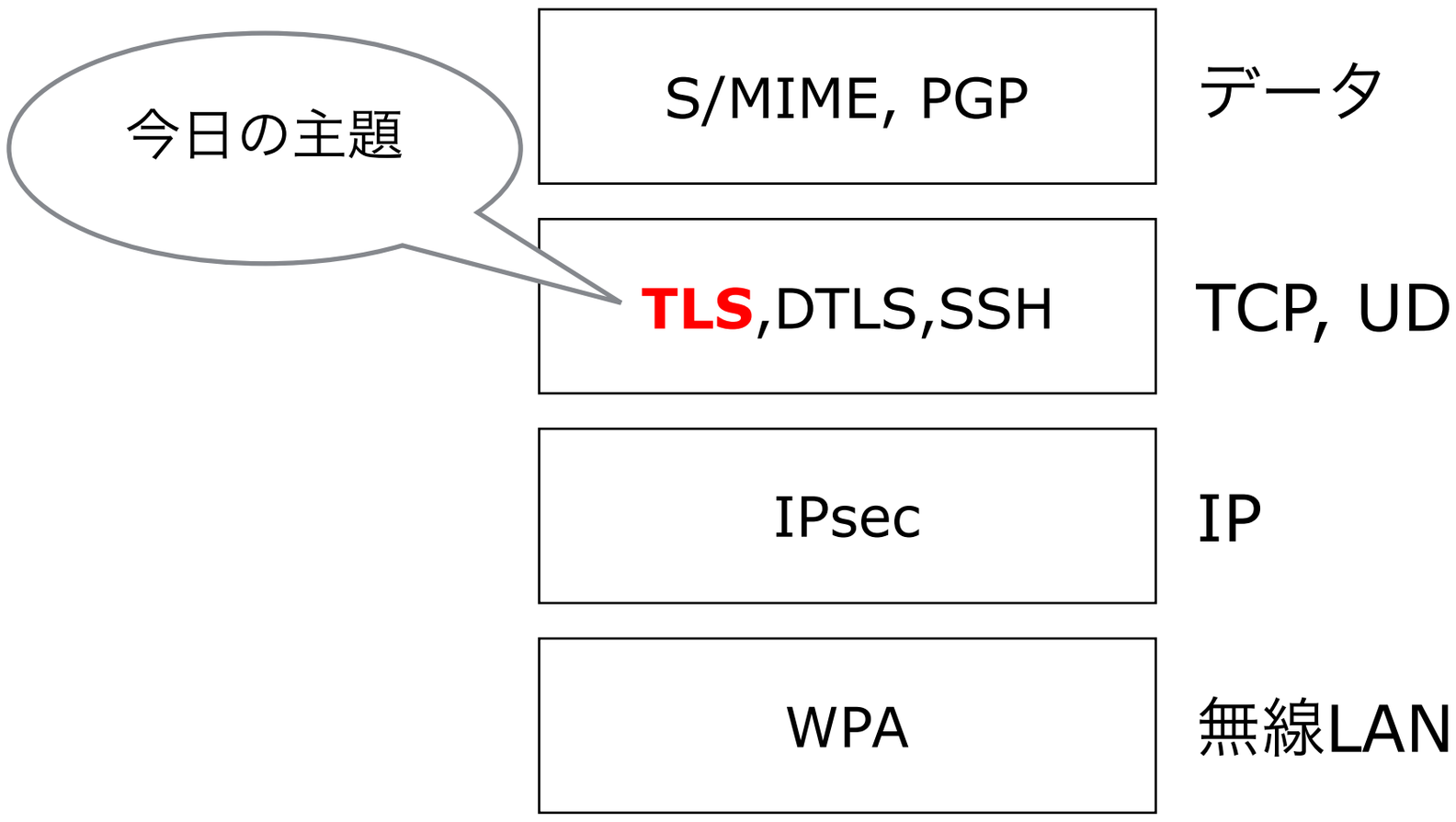
否認



インターネットの脅威から守るセキュリティ対策



各レイヤーにおけるセキュリティ通信



TLSの目的

RFC5246: The Transport Layer Security (TLS) Protocol Version 1.2

1. Introduction
The primary goal of the TLS protocol is to provide **privacy and data integrity** between two communicating applications.

- TLSプロトコルの最重要なゴールは、通信する２つのアプリケーションの間で**プライバシーとデータの完全性**を提供することです。

TLSの簡単な歴史

	SSL 1.0(未発表)
1994年	SSL 2.0 DROWN
1995年	SSL 3.0 POODLE
1996年	IETF TLS WG スタート
1999年	TLS 1.0 BEAST
2006年	TLS 1.1
2008年	TLS 1.2
2013年	TLS 1.3検討スタート
2016年	TLS 1.3仕様化完了？

SSLは旧ネットスケープ社の私的プロトコル

大人の事情で名称変更

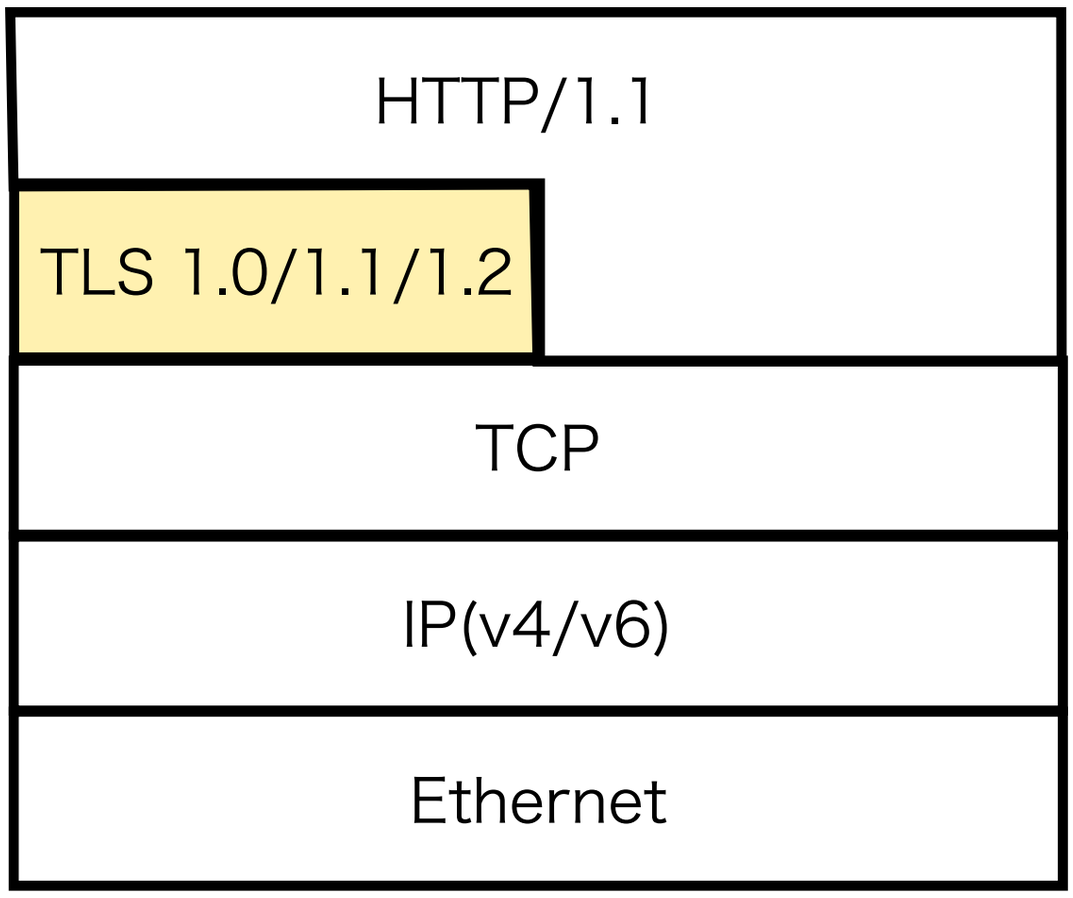
SSL3.0と基本設計は大きく変えず改良

様々な機能拡張

近日WGラストコールを目標
まだわかりません

TLSの位置付け

HTTP/1.1の時代

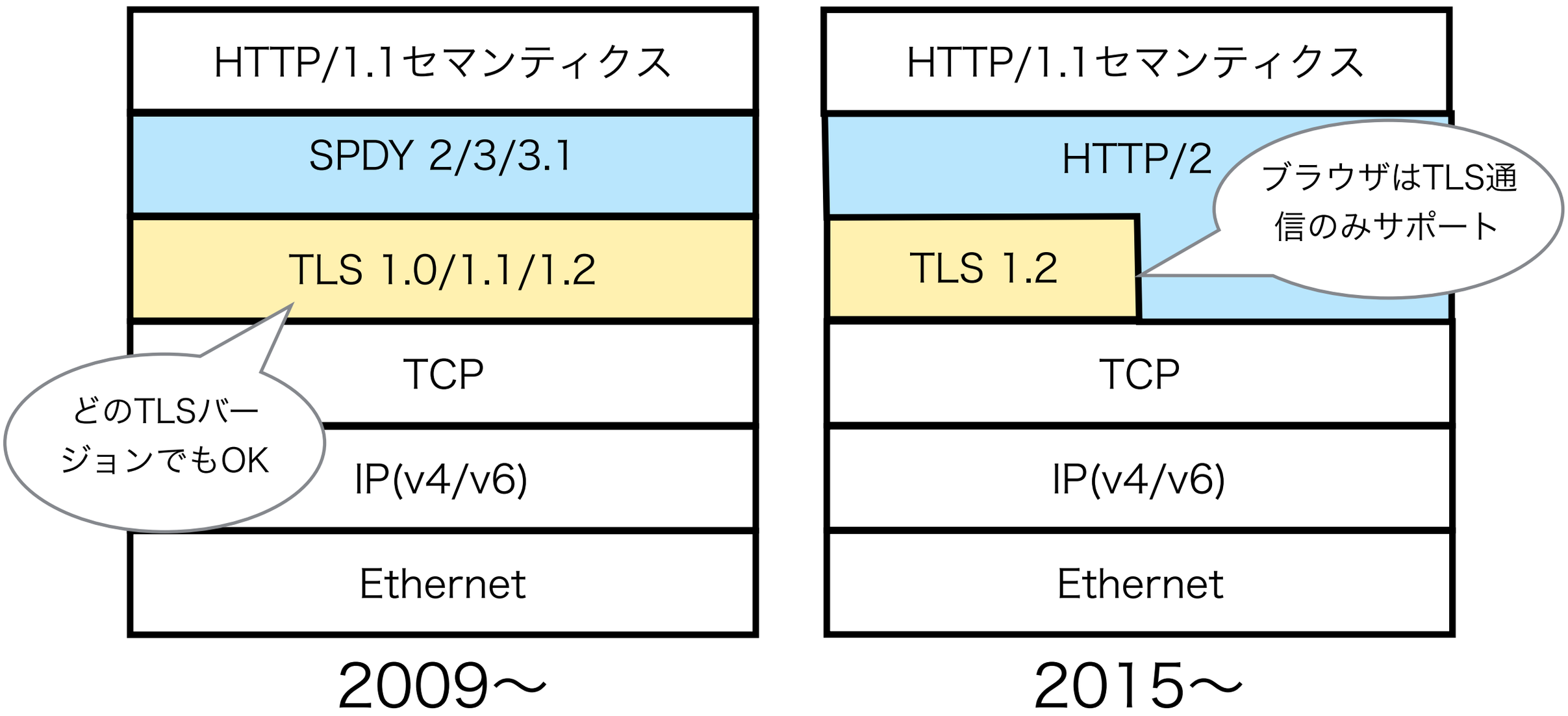


1999〜

(* TLS1.1 2006〜) (* TLS1.2 2008〜)

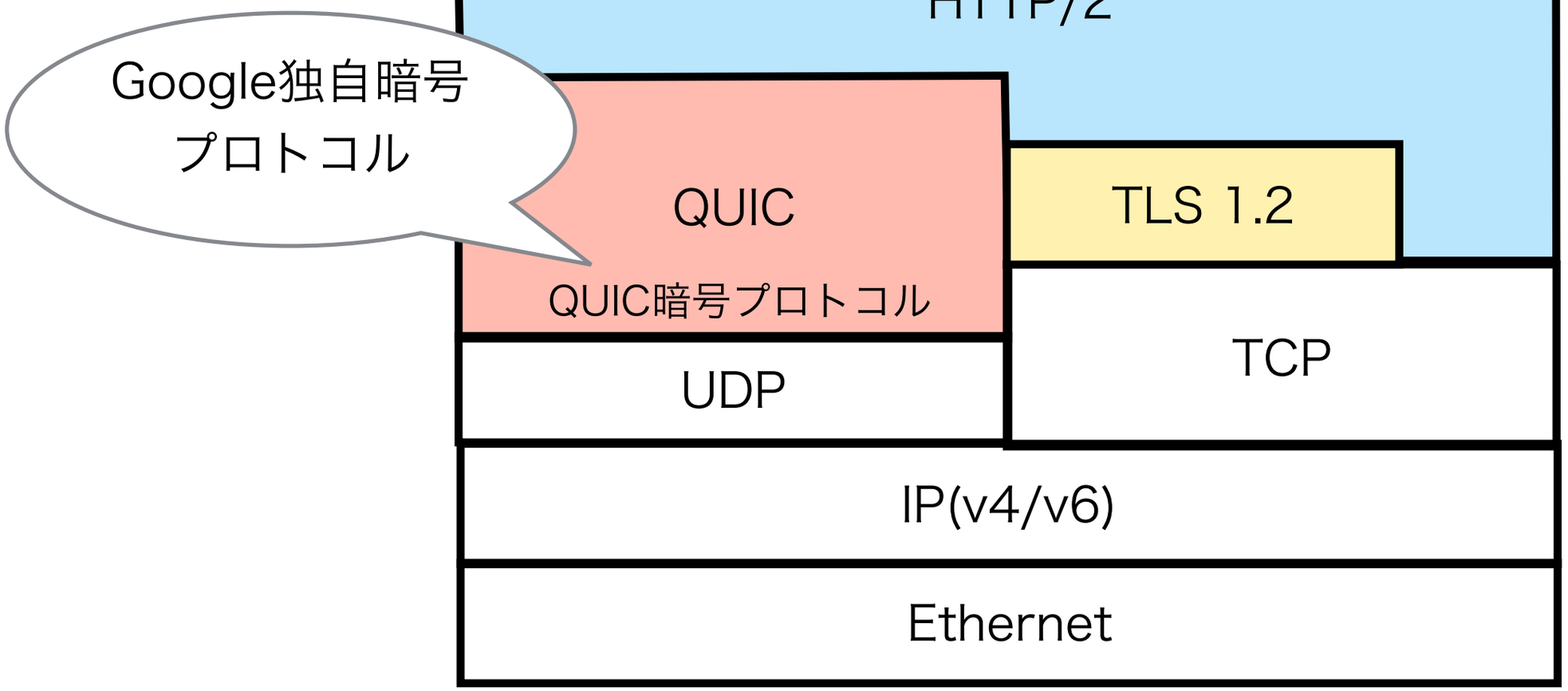
TLSの位置付け

HTTP/1.1からHTTP/2へ



TLSの位置付け

HTTP/2からQUICへ

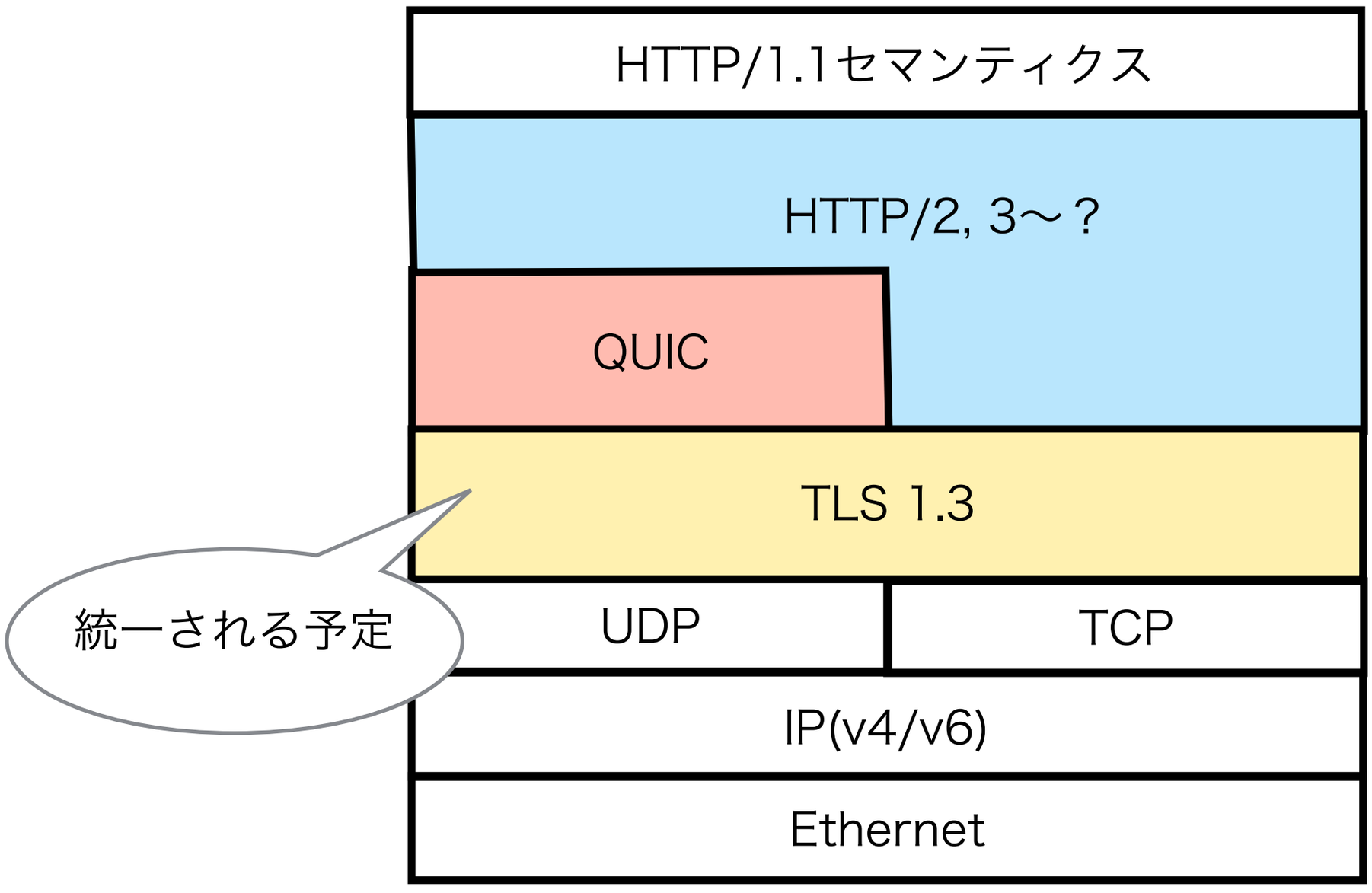


2013〜

(* HTTP/2 2015〜)

TLSの位置付け

QUICからTLS1.3へ



2017〜

なぜTLSが重要か？

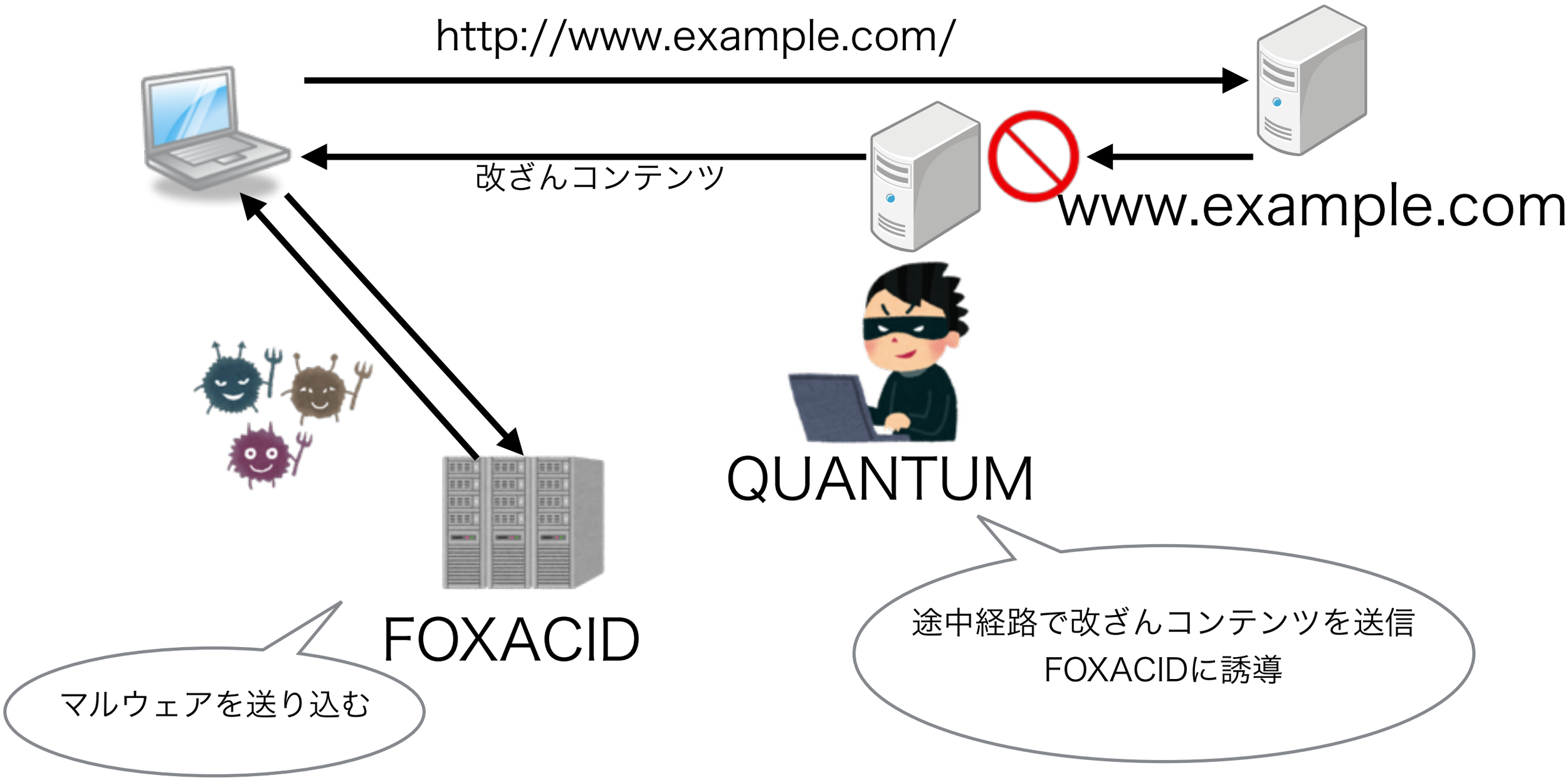
常時TLS時代の到来

Pervasive Surveillance
広範囲の盗聴行為

- 国家的な組織(米国NSAと英国GCHQなど)が莫大な予算で行う広範囲の盗聴行為
- 2013年6月 エドワード・スノーデンによってその活動内容がリークされる。

インターネット/電話の傍受・監視、データセンター内通信盗聴、暗号解読、暗号バックドア、サイバー攻撃 等

NSAによるサイバー攻撃の一例



https://www.schneier.com/blog/archives/2013/10/how_the_nsa_att.html

プロトコル技術者の憂慮

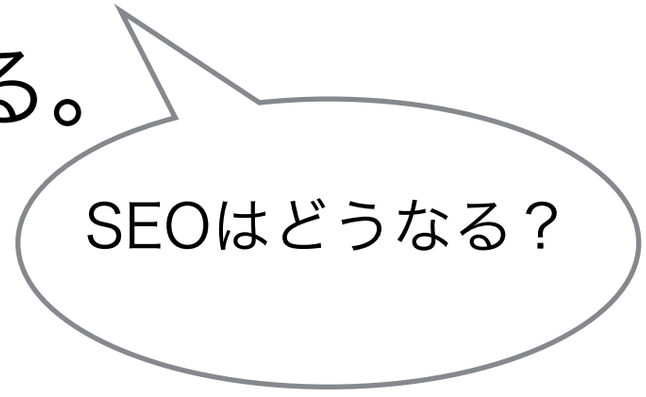
- 従来大規模な設備と予算が必要で現実的には無理と見られてきた攻撃が実際に行われていた。

- 公衆無線LANの普及など通信の盗聴・改ざんが可能な環境が広がってきている。

- 幸い最新の技術でしっかり暗号化された通信まではまだ破られておらず、安全であろう。

検索サービス会社の憂慮

- 検索のページランクが高いサイト宛の平文通信は、攻撃対象として当然狙われる。
- 平文通信でユーザがコンテンツ改ざんやマルウェア感染によってDDoS攻撃の一端を担う恐れもあり(Githubへの攻撃例)。
- ネットコンテンツの健全性の低下は、長期的に検索サービスへの信頼性を損なうことになる。



IAB(*)によるインターネットの信頼性に関する宣言(2014/11)

- 新しくプロトコルを設計する際には、**暗号化機能を必須**とすべき。

- ネットワーク運用者やサービス提供者に**暗号化通信の導入を推進**するよう強く求める。

- コンテンツフィルターやIDS等平文通信が必要な機能については将来的に代替技術の開発に取り組む。

(* Internet Architecture Board)
https://www.iab.org/2014/11/14/iab-statement-on-internet-confidentiality/

Mozillaによる
安全でないHTTPの廃止宣言

1. ある時期から新規機能は、HTTPSだけ利用できるようにする。

2. 現在HTTP(平文通信)で利用できる機能で、ユーザのセキュリティやプライバシーにリスクを与えるものを削除していく

https://blog.mozilla.org/security/2015/04/30/deprecating-non-secure-http/

ChromeのHTTP上の機能廃止

Chromeでは、下記の機能をHTTP(平文通信)で利用禁止する予定

- 位置情報を取得 (廃止済)
- デバイスの動きや方向を操作
- 暗号化された動画音声の再生
- カメラ・マイクなどの操作
- アプリケーションのキャッシュ情報の操作

https://sites.google.com/a/chromium.org/dev/Home/chromium-security/deprecating-powerful-features-on-insecure-origins

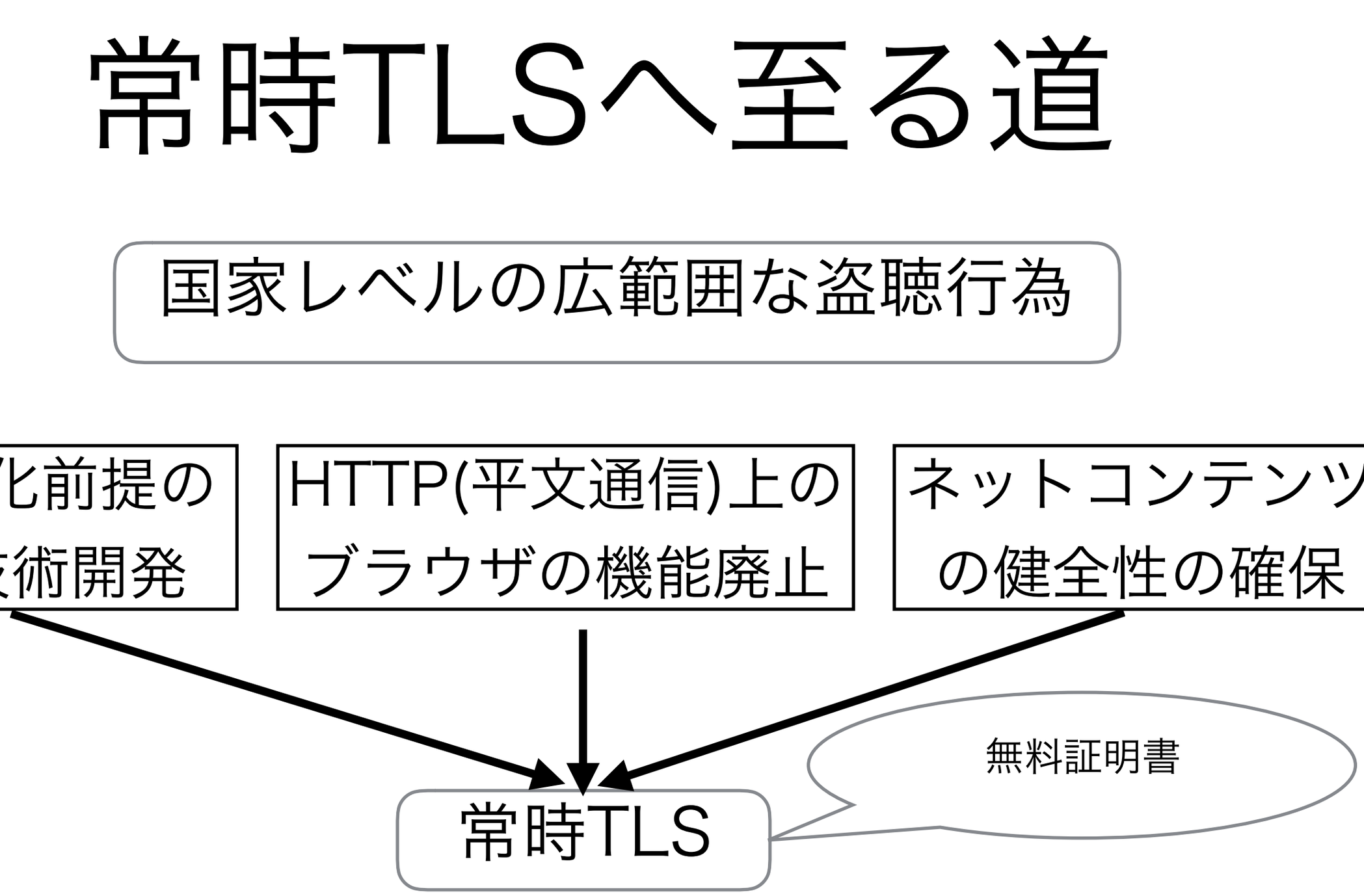
常時TLSへ至る道

国家レベルの広範囲な盗聴行為

暗号化前提の新技術開発

HTTP(平文通信)上のブラウザの機能廃止

ネットコンテンツの健全性の確保

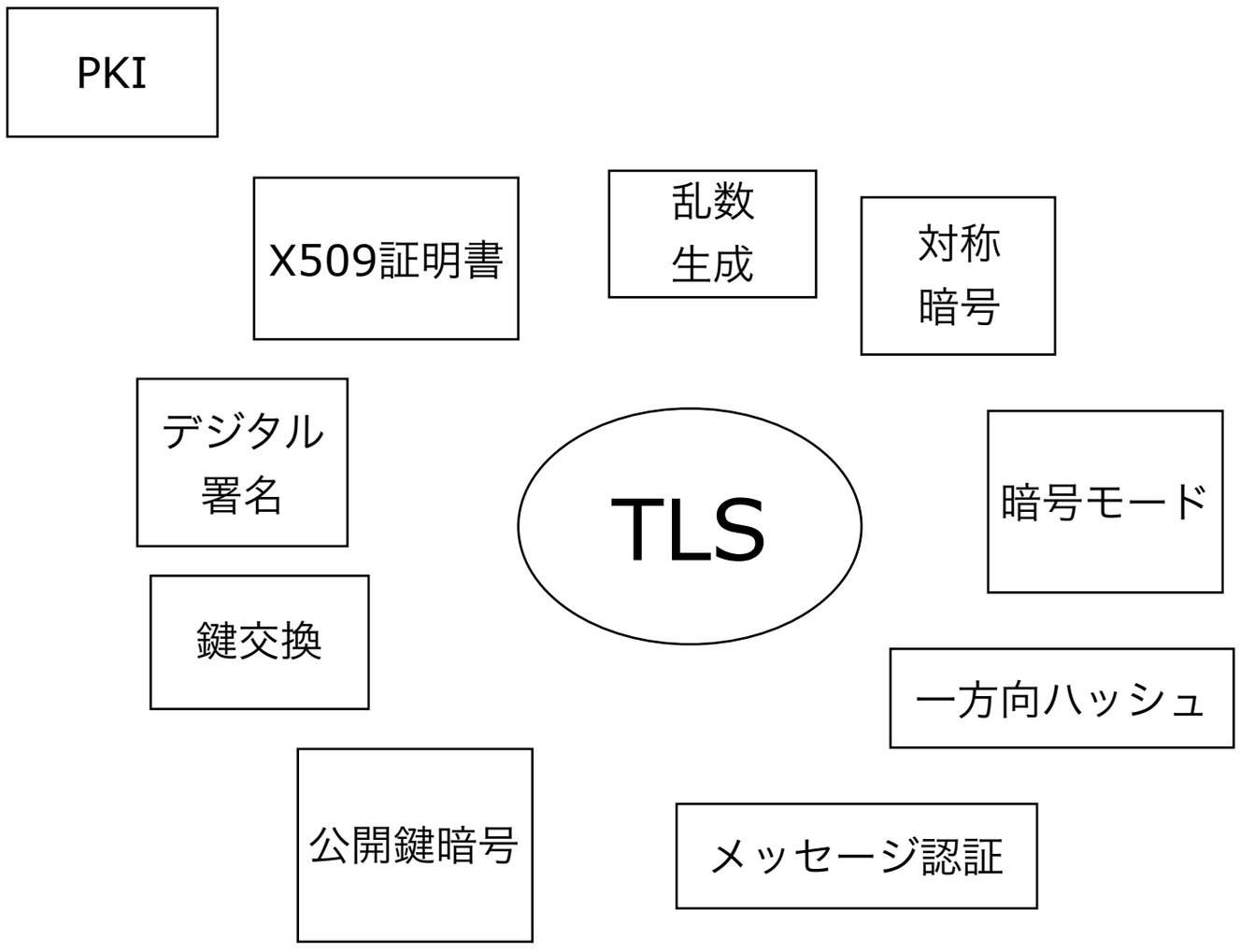


将来的な新技術はTLS利用を前提とする。

最先端の技術者はTLSを避けて通ることはできない。

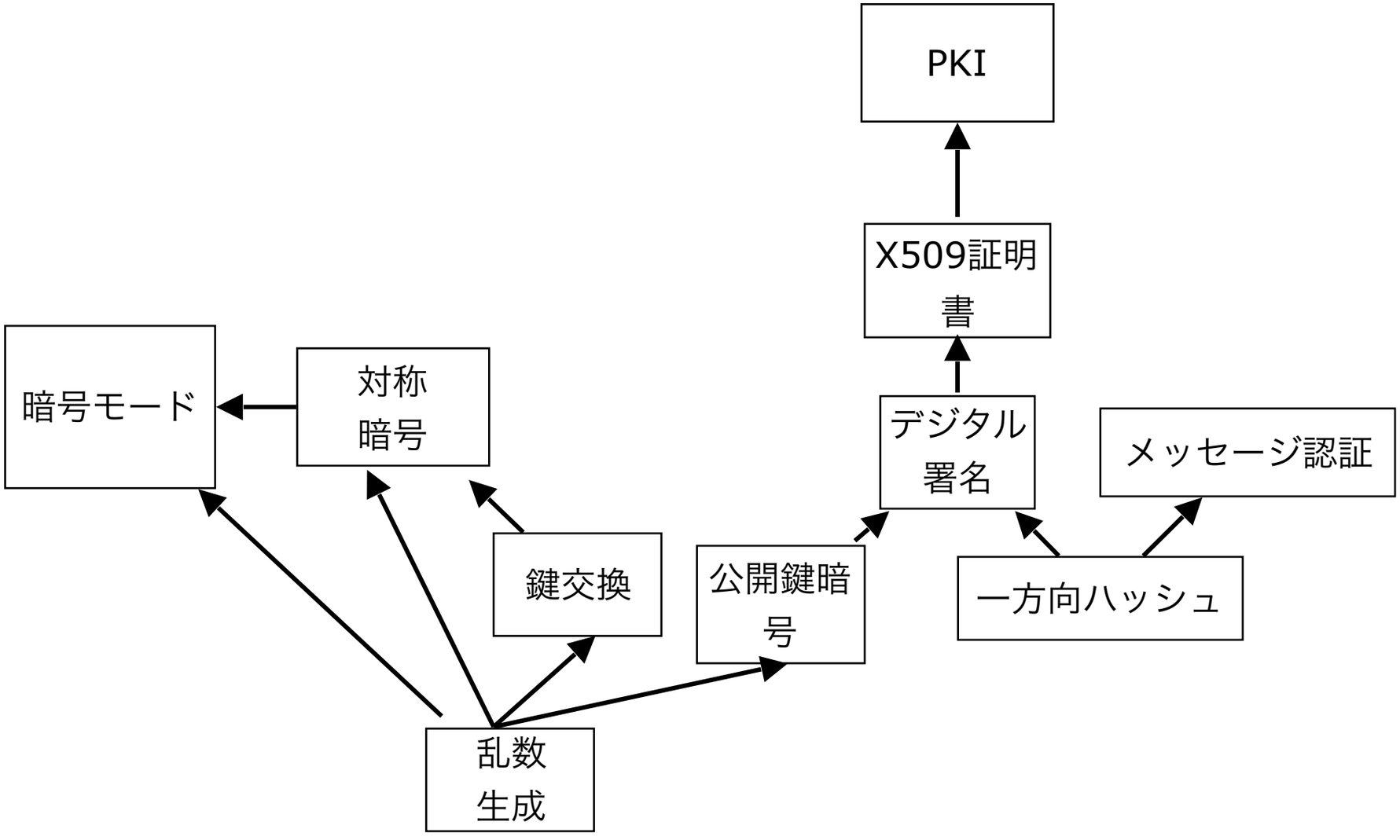
TLSを理解する準備

TLSの要素技術



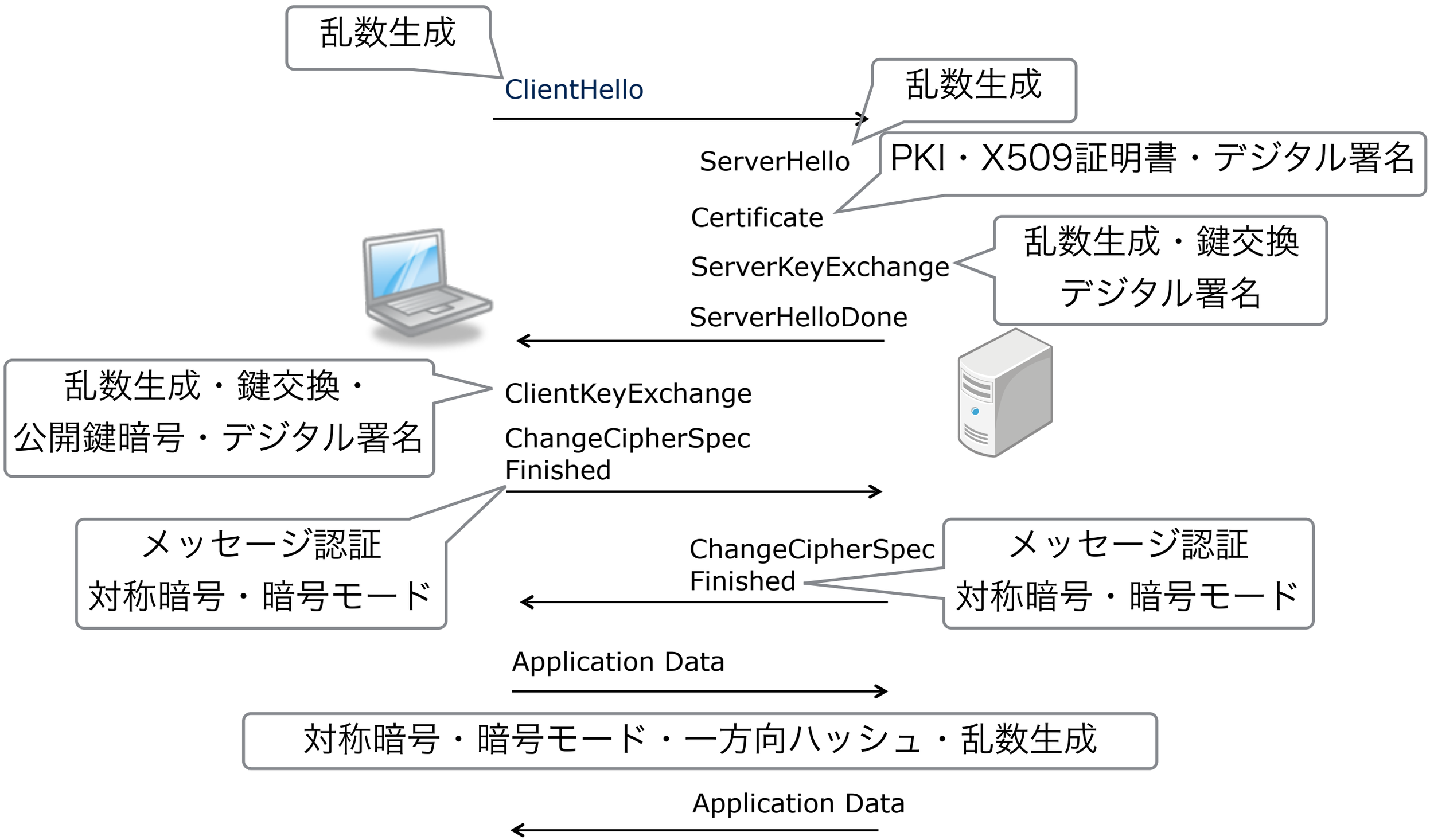
TLSプロトコルは、これらの要素技術を組み合わせて
アプリ間のセキュア通信を確立する手順を決める

TLS要素技術の依存性



本来はこの**一つ一つをきちんと理解すること**が必要

TLS要素技術はどこで使われる？

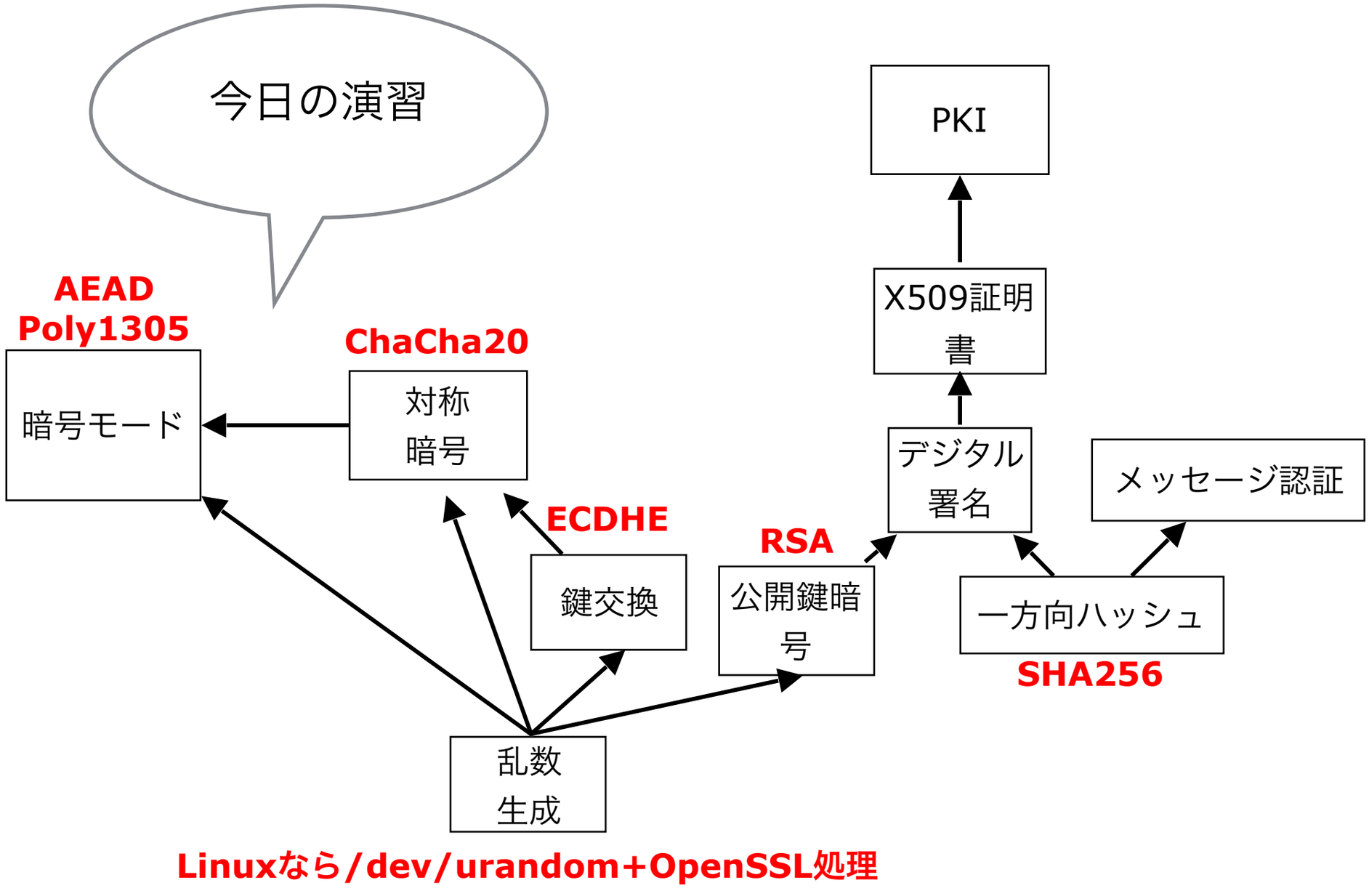


TLS要素技術はどこで使われる？

乱数生成	Client/ServerHelloのNonce, 鍵ペアの生成 データ暗号化のIV
PKI	CAによるサーバ証明書の署名と発行
X509証明書	Certificateによるサーバ・クライアントの認証・公開鍵の取得
電子署名	証明書の署名・鍵交換で交換する公開鍵の署名
鍵交換	Server/ClientKeyExchangeによる(EC)DH公開鍵の交換
公開鍵暗号	RSA鍵交換時にPreMasterSecretの暗号送信
一方向ハッシュ	CBCなどの暗号モード利用時にアプリデータのMAC生成
メッセージ認証	MasterSecretの生成、Finishedによるハンドシェイクデータの完全性検証
対称暗号・暗号モード	ChangeCipherSpec以降のハンドシェイクとアプリケーションデータの暗号化

(注：他にも細かいところで使われています。)

今回使うTLS要素技術



セットメニュー化されたTLSの要素技術

TLS CipherSuites

TLS _ 鍵交換 _ デジタル署名 _WITH_ 対称暗号 _ 鍵長 _ 暗号モード _ メッセージ認証(ハッシュ)

TLS_RSA_WITH_AES_128_GCM_SHA256 = {0x00,0x9C}

鍵交換・デジタル署名にRSA
対称暗号に128bit鍵長のAES
暗号モードにGCM(AEAD)
ハッシュにSHA256

TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256={0xCC,0xA8}

鍵交換にECDHE
デジタル署名にRSA
対称暗号にChaCha20
暗号モードにPoly1305(AEAD)
ハッシュにSHA256
番号として0xCC,0xA8を割り当て

今はTLSに何を使う？

鍵交換	RSA	Forward Secrecy			
		DHE	ECDHE		
デジタル署名	RSA	DSS (DSA)	ECDSA		
対象暗号	DES	RC4	AES	ChaCha20	その他
暗号モード	CBC	AEAD			
		CCM	GCM	Poly1305	
メッセージ認証（ハッシュ）	MD5	SHA-1	SHA256	SHA384	

ちなみに、
量子コンピュータで鍵交換、デジタル署名は全部アウト！

赤：使わない、　黄：注意、　緑：今のところ使って大丈夫

(注意は、暗号学的注意と将来的に普及が見込まれない注意も含まれます)

対称暗号

ストリーム暗号：データを逐次暗号化(RC4, Chacha20)

ブロック暗号：データをブロック毎に暗号化(DES, AES)

ブロック、ストリームの両者の違いは現在なくなってきている
ブロック暗号(AES)を暗号モード(後述)でカウンターモードを利用することにより全てストリーム暗号として利用できます。(AES-GCMはストリーム暗号処理)

幾つかの暗号では既に危殆化：

DES: 2005年 NIST FPS46-3規格の廃止(2030年までは許容)

RC4： RFC7455: Prohibiting RC4 Cipher Suites

対称暗号 AES

- 1997年よりプロジェクト開始、2000年選定、2001年仕様発行
- ブロックサイズ 128bit
- 鍵長： 128bits, 192bits, 256bits の３種類
- Intel/AMDのCPUでハードウェア処理のサポート(AES-NI)

２０１６年現在TLS通信のデファクト

ChaCha20は後でたっぷりと説明します。

暗号モード

- ブロック暗号は同じデータを同じ鍵で暗号化すると毎回同一の暗号文になる。
- ブロック長より長いデータを暗号化する場合に暗号モードを利用して繰り返しを避ける。

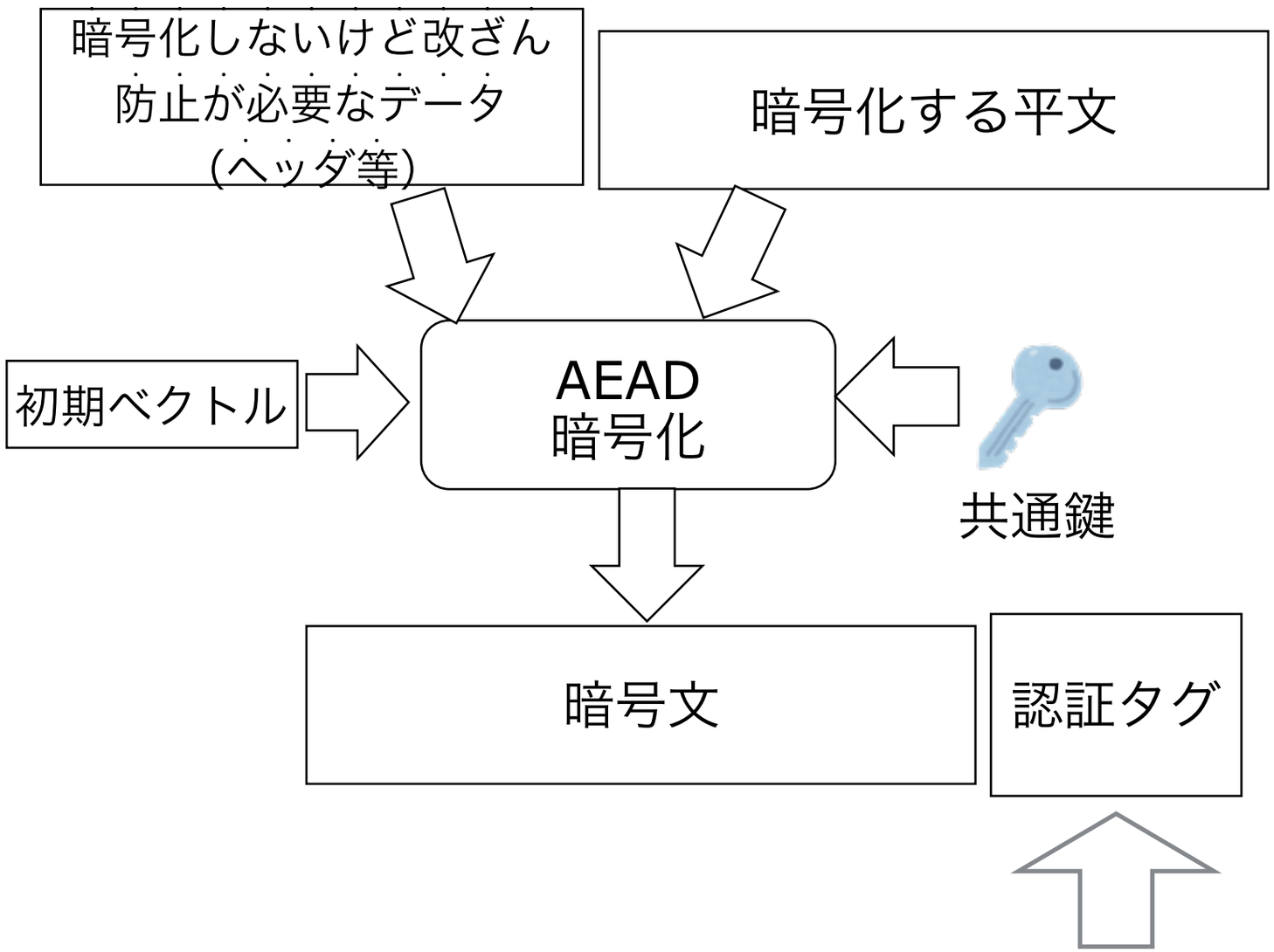
これまでの主流

- CBC：「(平文 XOR ベクトル) を暗号化」を続ける
- CTR：「カウンターを暗号化 XOR 平文」を続ける

実際にTLSで利用するには改ざん検知のためのMAC(メッセージ認証）との組み合わせる(AEAD)。AES-GCMが今の主流。

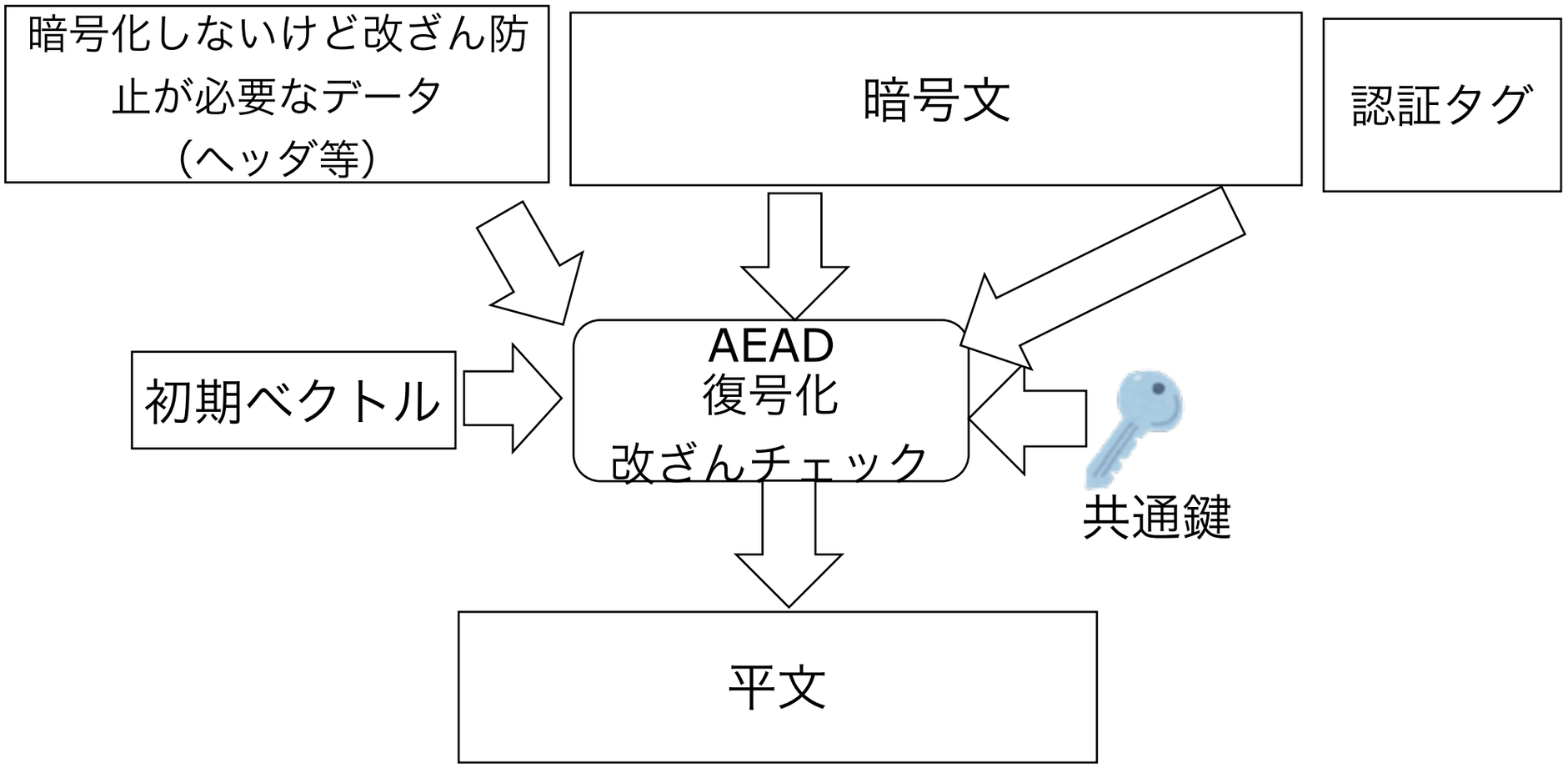
ChaCha20-Poly1305は後でたっぷりと説明します。

AEAD（認証付き暗号）



Encrypt-Then-MAC(暗号化した後でハッシュ値を取得)

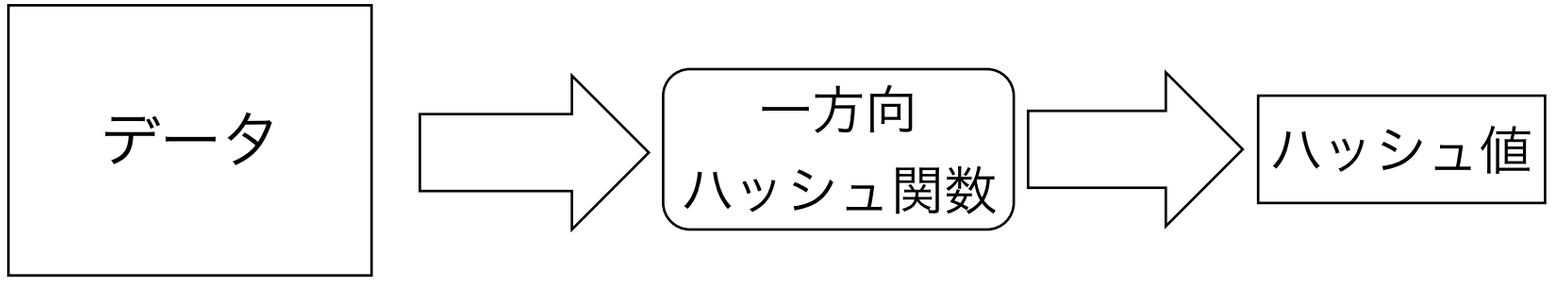
AEAD（認証付き暗号）


暗号化しないけど改ざん防止が必要なデータ（ヘッダ等）
暗号文
認証タグ
初期ベクトル
AEAD
復号化
改ざんチェック
共通鍵
平文

GCM

- GCM (Galois Counter Mode: ガロアカウンターモード)
- CTRとGHASHを組み合わせたAEAD
- ハードウェア処理で高速化が可能
- AESと組み合わせて AES-GCMとして利用

一方向ハッシュ

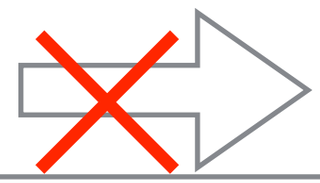


ハッシュ値を比較することでデータの改ざんをチェックすることができる。

暗号学的ハッシュ

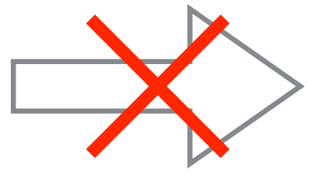
・原像計算困難性(Preimage Resistance)

ハッシュ値hからもとのメッセージmを探すのが困難

h  h == HASH(m)のmを見つける

・第２原像計算困難性(2nd-Preimage Resistance)

特定のメッセージm1と同じハッシュ値を持つm2を探すのが困難

h1 = HASH(m1)  HASH(m2) == h1のm2を見つける

・強衝突耐性(Strong Collision Resistance)

HASH(m1) == HASH(m2)となるm1とm2を見つけるのが困難

一方向ハッシュ

- md5

既に現実的な攻撃手法が存在

- SHA-1

2018年ぐらいには現実的なコストで衝突データを探せる見込み(*2)

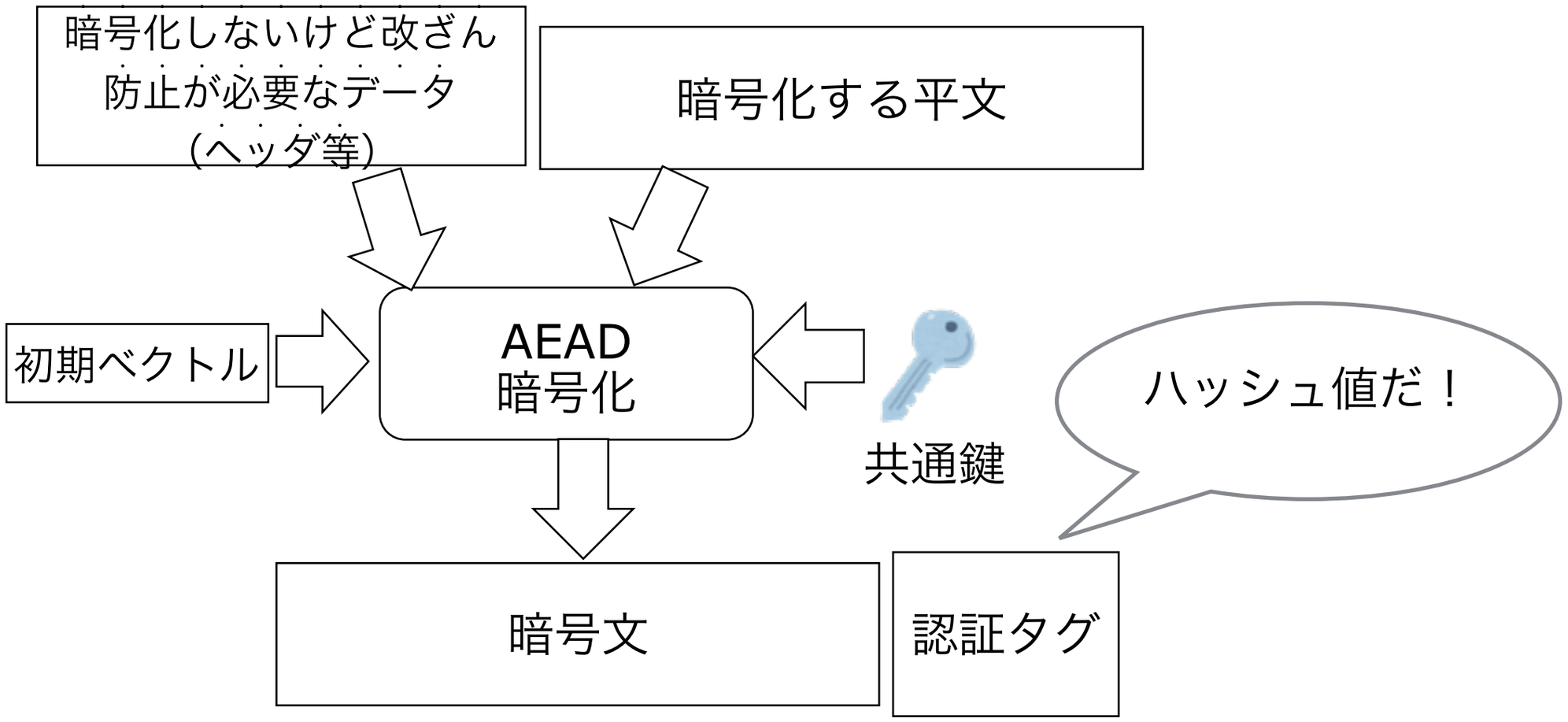
- SHA-2(SHA-256など6種)
- SHA-3(SHA3-256など6種)

8/5にNISTより正式公開

(*1) how to Break MD5 and Other Hash Functions
http://merlot.usc.edu/csac-f06/papers/Wang05a.pdf

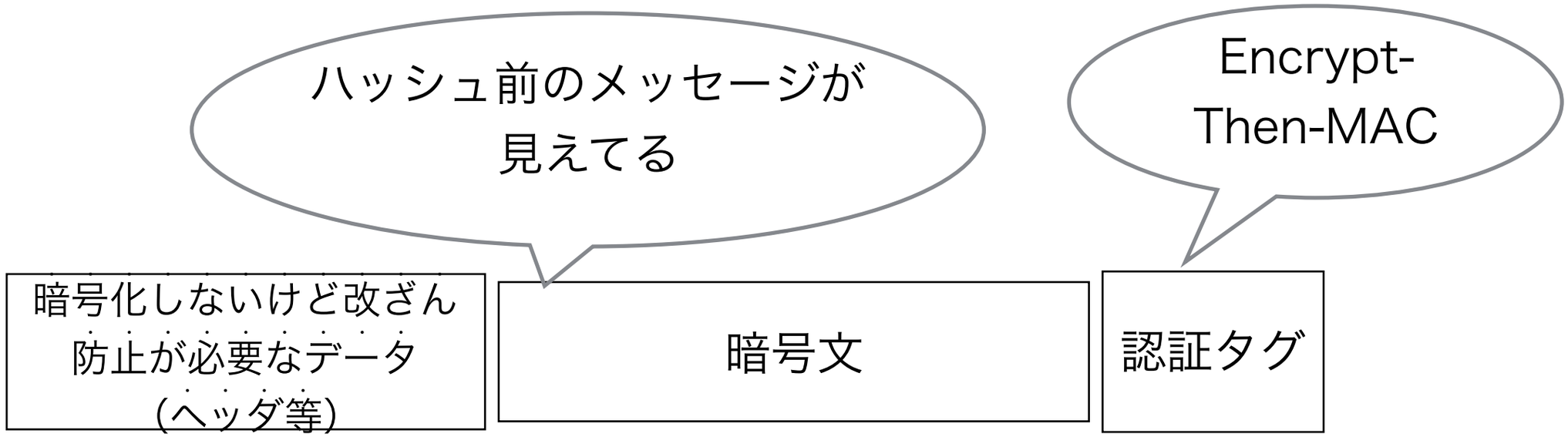
(*2) Cryptanalysis of SHA-1
https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

AEADを思い出そう



GCM, Poly1305: あれっ、SHA256とかじゃない。なぜ？

AEADでは暗号学的ハッシュまでは必要ない



原像計算困難性はいらない。

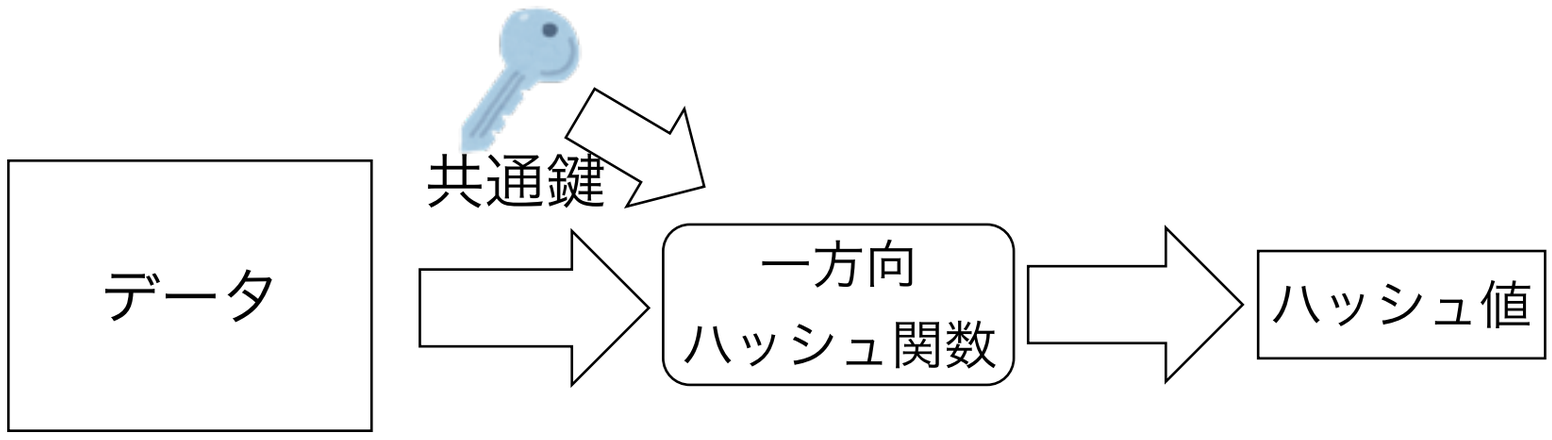
見えてるメッセージの改ざん検知(MAC)が重要。

第２原像計算困難性と高い強衝突耐性が求められる。

パケット毎に計算するので高速性能大事。

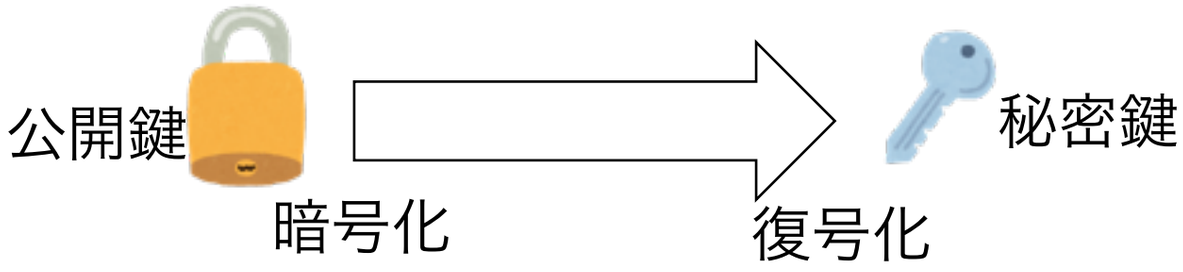
GCM/Poly1305は、AEAD向けに特化した高速MACアルゴリズム

メッセージ認証(HMAC)



- 事前に共通鍵を共有
- 共通鍵とデータを組み合わせたハッシュ値を作成
- データの完全性とハッシュ作成者を認証する

公開鍵暗号

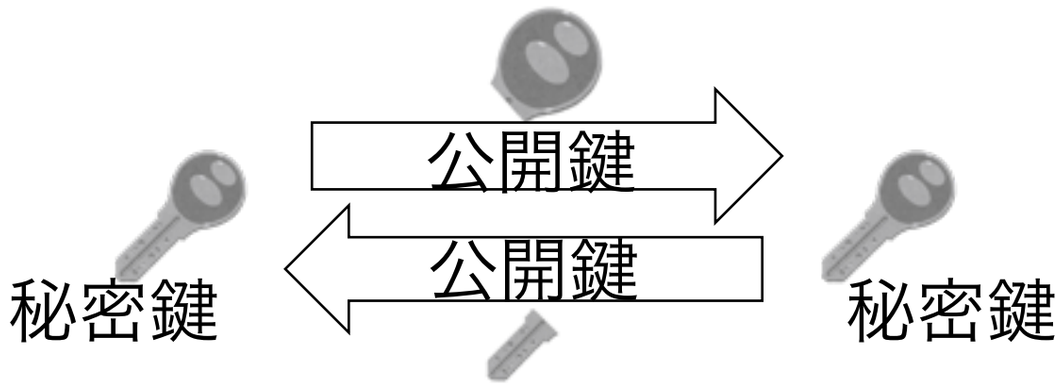


- 解を求めるのが困難な数学的問題を利用して暗号を生成。
- 公開鍵と秘密鍵のペアを生成。公開鍵はさらして大丈夫。
- 公開鍵で暗号化し秘密鍵で復号化。

- RSA 素因数分解
- ECC(楕円曲線暗号）楕円曲線上の離散対数問題

512bit RSAの危険性 FREAK https://freakattack.com/

鍵交換



- 2者間で安全に鍵を共有する仕組み
- 互いに公開鍵を交換しあい、共有鍵を生成する。
- 通信経路上で共有鍵のやり取りがない。

一時的な鍵交換はE(Ephemeral)の文字が付く

- DH (Diffie-Hellman) → DHE
- ECDH(楕円曲線DH) → ECDHE

脆弱性：DH Logjam https://weakdh.org/

デジタル署名

- データの完全性のチェックが可能となる。
- データの送信元の認証が可能となる。
- 公開鍵の信頼性の範囲で否認防止が可能となる。

- RSA
- DSA,ECDSA

PKI概要

論理的に複数の役割に分かれているが物理的に１つでもよい

CA
(Certificate Authority)

RA
(Registration Authority)

VA
(Validation Authority)

CRL/OCSP

OS・ブラウザ
ベンダー

サーバ証明書

CSR **実在確認**

Root証明書

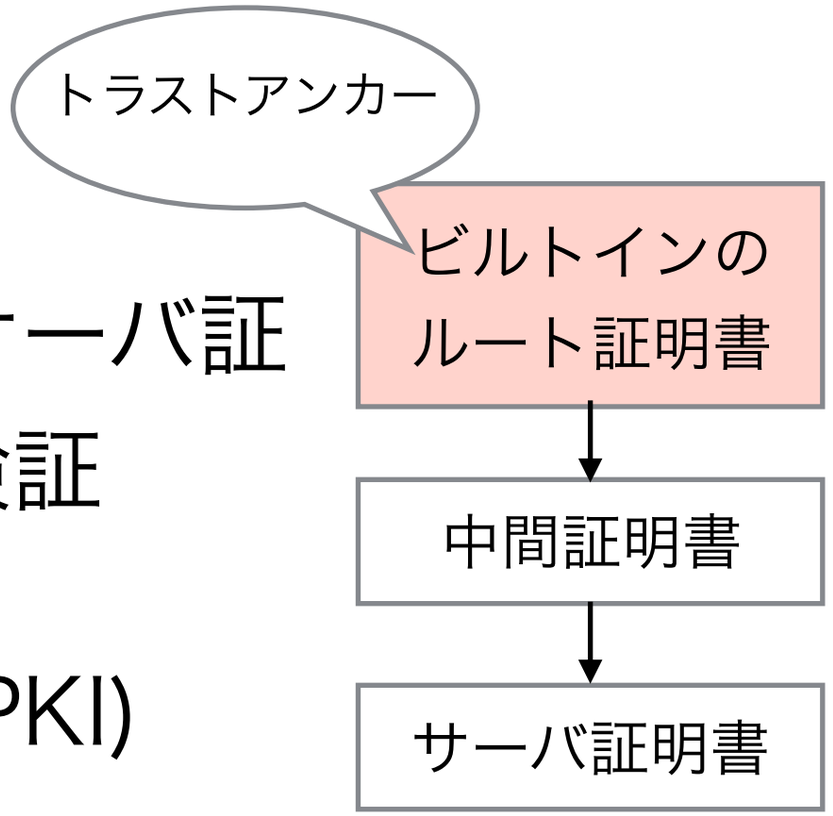
失効確認

鍵ペア

https://～

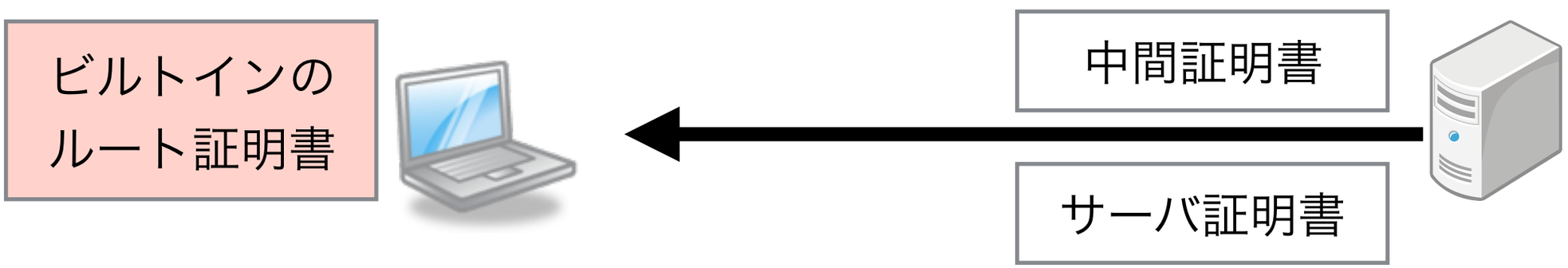
サーバ証明書(X509)

- TLS通信の信頼性を担保する要



- ビルトインのルート証明書からサーバ証明書まで証明書チェーンの署名検証

- オンライン以外で信頼性を担保(PKI)



証明書の種類

ネットワーク以外の実在証明

EV証明書 (Extended Validation)	CA共通の厳格な組織の実在証明 (物理的実在, 書面やデータ, 口座取引による実在審査・署名提出・電話確認など) アドレスバーが緑色
OV証明書 (Organization Validation)	各CAポリシー(CPS)に従った組織の実在証明 (書面やデータ審査・電話確認など)
DV証明書 (Domain Validation)	各CAポリシー(CPS)に従ったドメイン保持証明 (メールの到達性確認など)

Let's Encrypt など
無料証明書があるよ

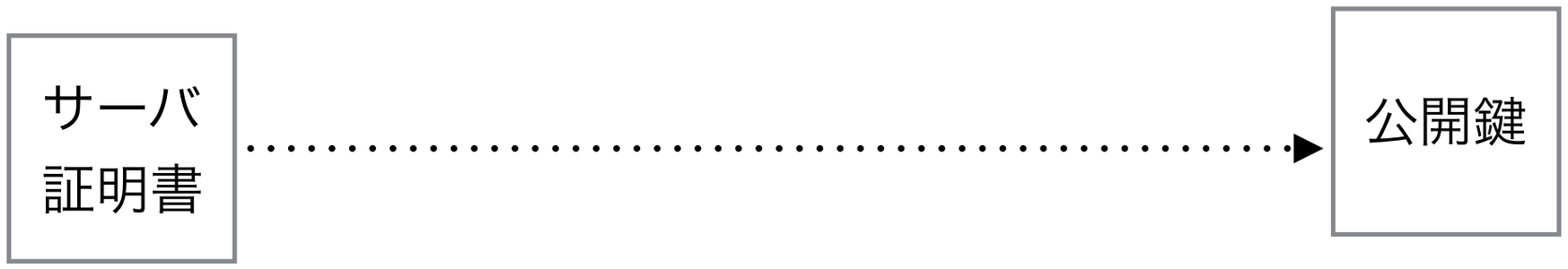
サーバ証明書の中身

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            60:77:fb:85:03:96:10:06:6c:62:ca:8a:f2:76:68:06
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=CN, O=WoSign CA Limited, CN=WoSign CA Free SSL Certificate G2
        Validity
            Not Before: Nov 27 07:13:30 2015 GMT
            Not After : Nov 27 07:13:30 2016 GMT
        Subject: CN=server21.hokkaido.koulayer.com
        Subject Public Key Info:
```

バージョン、シリアル番号、発行者情報、有効期限、サーバ識別子、公開鍵情報、拡張情報(利用用途、別名や失効情報・ポリシー参照先)、デジタル署名

サーバ証明書の確認

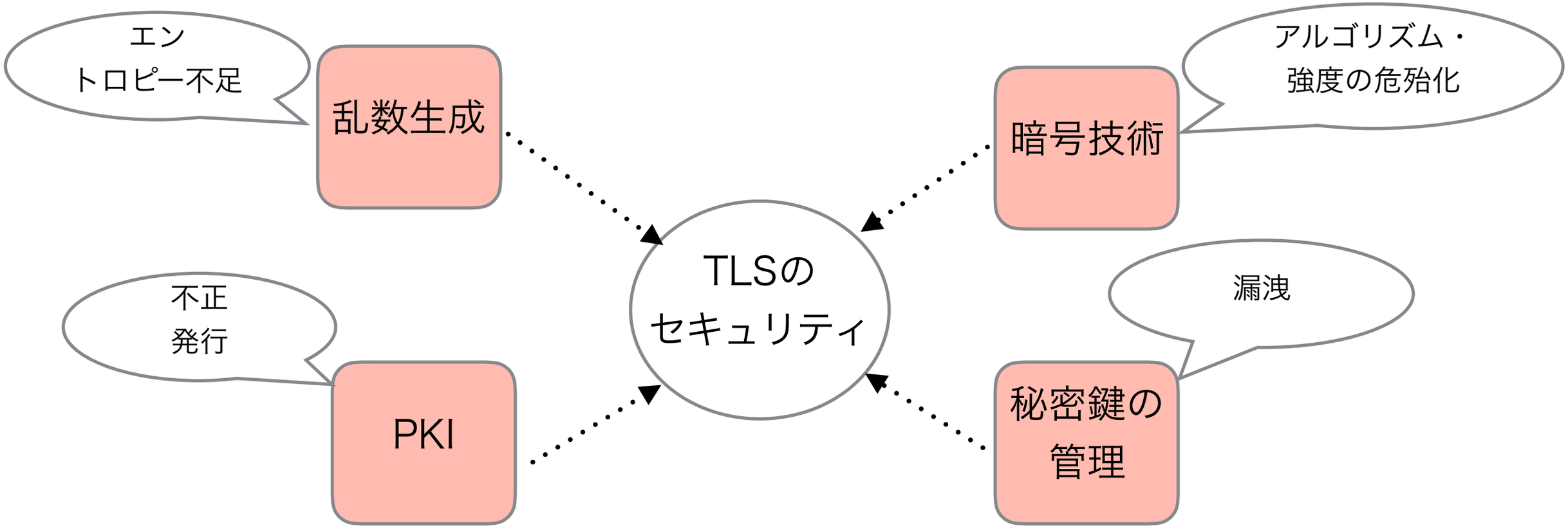
サーバ証明書と秘密鍵の対応が間違っていたらTLSサーバは起動しない。なのでサーバ証明書と秘密鍵の公開鍵が一致するか必ずチェックする。



openssl x509 -pubkey -inserver.crt -noout > server_pubkey.pem

openssl rsa -pubout -in private.key -out private_pubkey.pem

TLSセキュリティの土台

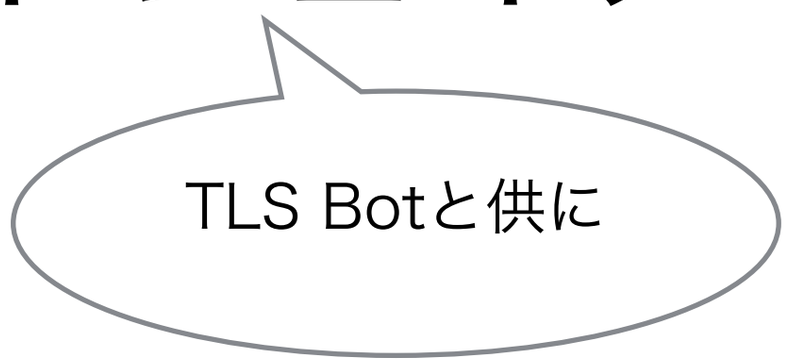


TLSは、この４つの外部要素の上でインターネットで安全な通信を提供する仕組みである。

逆に言えば、どれほど完璧なTLSプロトコルを作っても

この４つの外部要素が破られたら安全を確保できない。

TLSハンドシェイク



注：複雑さを避けるためクライアント認証機能の説明は省略します。

演習

実際に ChaCha20 のパケットを見てみる

https://chacha20.tls.koulayer.com/

に Chromeでアクセス、Developer Toolで確認してみる。

https://chacha20.tls.koulayer.com/chacha20_sample.pcap
をダウンロードして、Etherreal で見てみよう。

SecCamp2016 TLS Bot

- コマンドラインでHEX形式のTLSフレームを入力してTLSハンドシェイクを行うBot

- Server/Client両方で動きます。

- Clientは最初にHelloRequestのフレームを入力して開始。

- NODE_DEBUG=seccamp2016 で出力フレームのJSONを出力します。

SecCamp2016 TLSBot

- npm install seccamp2016-tls-exercise

- Server/Client Botのスクリプトを作成

```
const SecCamp2016 = require('seccamp2016-tls-exercise');

SecCamp2016.TLSBot(false); // client は false
```

インストールされた node_module が見つかれば
node_modules/seccamp2016-tls-exercise/samples/
にコードがあります。

https://gist.github.com/shigeki/8f116a8689a60b3cbbf9ed8618fae8ba
にもあります。

TLS Bot

1. screen

```
ohtsu@omb:seccamp2016$ node tls_client_bot.js
TLSBot only supports TLS1.2(ECDHE-RSA-CHACHA20-POLY1305 with prime256v1 and sha256 sig.)
Start with HelloRequest: 1603030004000000000
TLS Client> 1603030004000000000
<= HelloRequest
ClientHello => 160303004501000041030 3f273f7f623a674a660ef854577132d35f48c3cbab18709401005
0016000b00020100000a00040002001 7000d000400020401
TLS Client>
```

1. screen

```
ohtsu@omb:seccamp2016$ node tls_server_bot.js
TLSBot only supports TLS1.2(ECDHE-RSA-CHACHA20-POLY1305 with prime256v1 and sha256 sig.)
TLS Server> 160303004501000041030 3f273f7f623a674a660ef854577132d35f48c3cbab18709401005cf
6000b00020100000a00040002001 7000d000400020401
<= ClientHello
ServerHello => 1603030032020000 2e03037955db3e56bbdf3b466a667c5c52c48e0dda4d3d80feb1851ba
0b00020100
Certificate => 16030302fa0b0002f60002f30002f0308202ec308201d4020203e7300d06092a864886f70
550406 13024a50310e300c06035504080c05546f6b796f3113301106035504070c0a436869796f64612d6b75
```

TLS bot Debugモード

export NODE_DEBUG=seccamp2016

```
ohtsu@omb:seccamp2016$ export NODE_DEBUG=seccamp2016
ohtsu@omb:seccamp2016$ node tls_client_bot.js
TLSBot only supports TLS1.2(ECDHE-RSA-CHACHA20-POLY1305 with prime256v1 and sha256 sig.)
Start with HelloRequest: 1603030004000000000
TLS Client> 1603030004000000000
<= HelloRequest
SECCAMP2016 51091: { ContentType: <Buffer 16>,
  ProtocolVersion: <Buffer 03 03>,
  Length: <Buffer 00 45>,
  Handshake:
   { HandshakeType: <Buffer 01>,
     Length: <Buffer 00 00 41>,
     ProtocolVersion: <Buffer 03 03>,
     Random: <Buffer 4f e3 d6 88 e2 33 74 4c 34 b4 66 97 f6 e1 b2 9d d4 d2 00 b8 13 bf 34 e5 f7 e8 da 63 1b 58 4d d
1>,
     SessionID: <Buffer >,
     CipherSuites: [ <Buffer cc a8> ],
     CompressionMethods: <Buffer 00>,
     Extensions:
      [ { Type: <Buffer 00 0b>, Data: <Buffer 01 00> },
        { Type: <Buffer 00 0a>, Data: <Buffer 00 02 00 17> },
        { Type: <Buffer 00 0d>, Data: <Buffer 00 02 04 01> } ] },
  remaining_buffer: <Buffer > }
ClientHello => 160303004501000041030 34fe3d688e233744c34b46697f6e1b29dd4d200b813bf34e5f7e8da631b584dd1000002cca80100
0016000b00020100000a00040002001700 0d000400020401
```

TLSハンドシェイク(full handshake)

ClientHello

ServerHello
Certificate
ServerKeyExchange
ServerHelloDone

ClientKeyExchange
ChangeCipherSpec
Finished

ChangeCipherSpec
Finished

Application Data

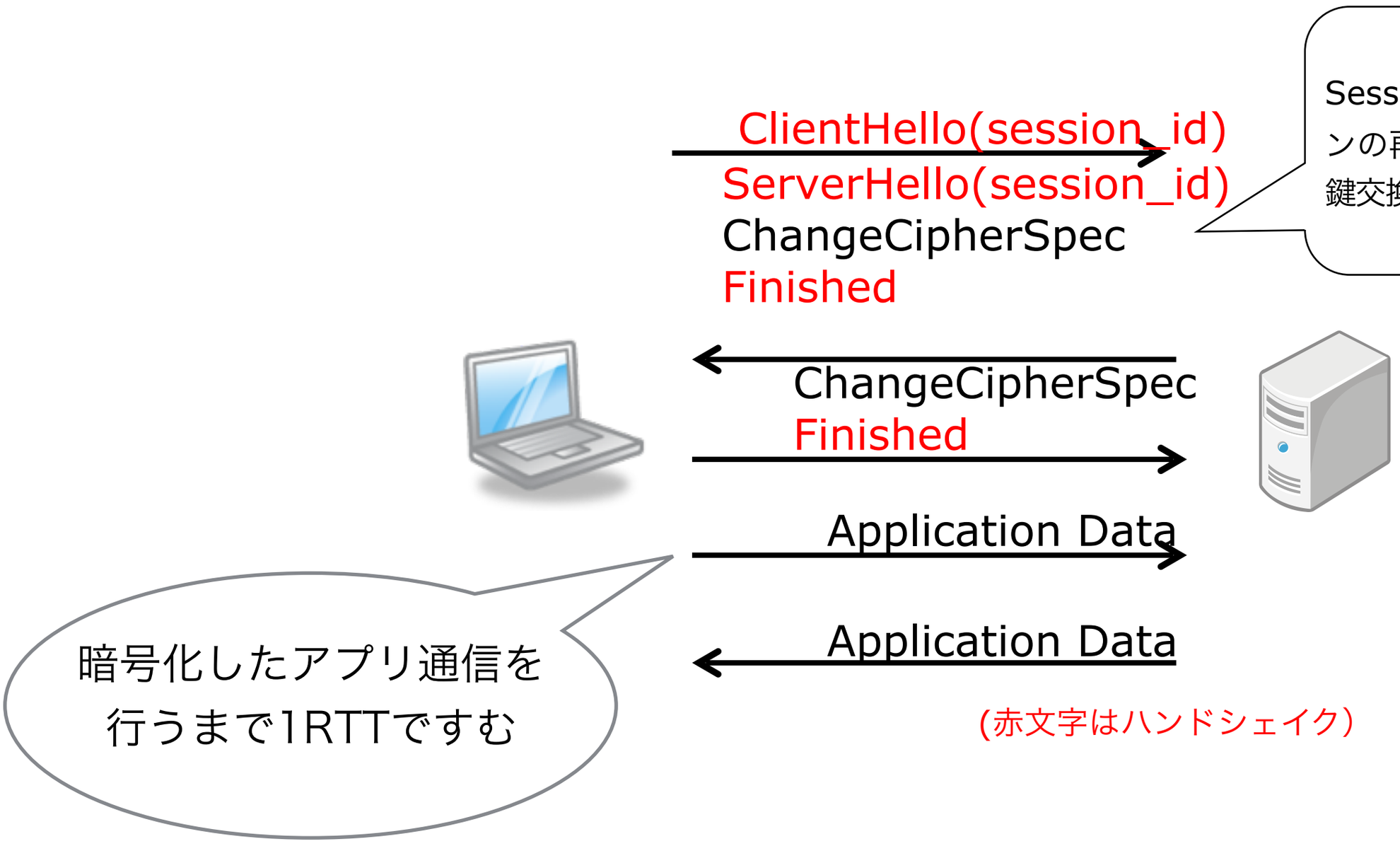
Application Data

ClientHelloとServerHelloのやり取りで双方が利用するTLSバージョンや暗号化方式などを合意する。

暗号化したアプリ通信を行うまで2RTT必要

(赤文字はハンドシェイク)

TLSハンドシェイク(resumption)



今回は演習の対象外です

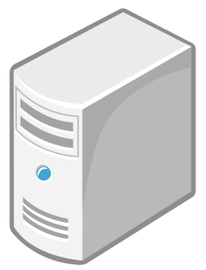
TLSハンドシェイクの意味

ClientHello/ServerHello/ServerHelloDone
TLSのための情報交換

バージョン・乱数・暗号方式・拡張情報

Certificate
公開鍵情報の送付

エンドポイントの認証



ClientKeyExchange/ServerKeyExchange
共有鍵交換

ChangeCipherSpec
暗号開始の合図

Finished
ハンドシェイクデータの改ざんチェック

TLS1.2の構造

IPヘッダ	TCPヘッダ	TLS Record Layer (5バイト)		
		タイプ（４種類）(1byte)	バージョン (2byte)	長さ (2byte)

ChangeCipherSpec (タイプ:0x14)
タイプ

Alert (タイプ:0x15)	
レベル	理由

Handshake (タイプ:0x16)		
msgタイプ（１０種類）	長さ（3バイト長）	ハンドシェイクデータ

TLS Record Layerデータに続いて、次の４**種類**のTLSデータのいずれかが続く。

Application Data (タイプ:0x17)
暗号化されたデータ

TLS Handshakeは、この**１０種類**に分かれる。

msgタイプ	ハンドシェイクデータの種類
0x00	HelloRequest
0x01	ClientHello
0x02	ServerHello
0x0b	Certificate
0x0c	ServerKeyExchange
0x0d	CertificateRequest
0x0e	ServerHelloDone
0x0f	CertificateVerify
0x10	ClientKeyExchange
0x14	Finished

TLSハンドシェイクフレームを読む

```
ohtsu@ohtsu-mac:seccamp2016$ node tls_client.bot
TLSBot only supports TLS1.2(ECDHE-RSA-CHACHA20-P
Start with HelloRequest: 160303000400000000
TLS Client> 160303000400000000
<= HelloRequest
ClientHello => 160303004501000041030 3b37686012c2
000200170000d000400020401
TLS Client>
```

5バイト　　4バイト

Record Layer					**Handshake (ClientHello)**						
type	protocol version		length (2byte)		msg type	length (3byte)			client version		random
	major	minor							major	minor	
0x16	0x03	0x03	0x00	0x45	0x01	0x00	0x00	0x41	0x03	0x03	32 byte

暗号化されない　　暗号化される

演習

- ２つのコマンドラインターミナルを開いて一つは tls_client_bot、もう一つは tls_server_bot を起動する。

- tls_client_bot に HelloRequestを入力して、出力したClientHelloをコピーして server bot に入力しよう

- NODE_DEBUG=seccamp2016 の設定をして JSON を確認しよう。

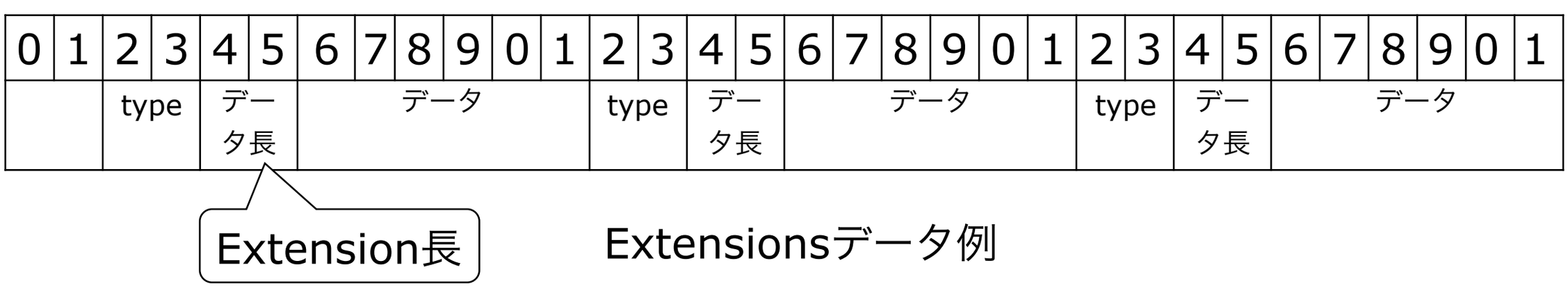
ClientHello

ClientHello

ClientHelloとServerHelloの
やり取りで双方が利用するTLS
バージョンや暗号化方式など
を合意する。

ClientHello

項目	要素	サイズ	先頭の長さ情報
client_version	uint8 major, uint8 minor	2	N/A
random	uint32 gmt_unix_time, opaque random_bytes[28]	4 + 28	N/A
session_id	opaque SessionID	<0..32>	1バイト分
cipher_suites	uint8 CipherSuite[2]	<2..2^16-2>	2バイト分
compression_methods	null(0)	<1..2^8-1>	1バイト分
extensions	extension_type(65535), extension_data<0..2^16-1>	<0..2^16-1>	2バイト分



Extensionsデータ例

ClientHello

クライアントが利用できる最高のTLSバージョンを指定、サーバがどのバージョンを使うか選択する

Record Layer			**Handshake (ClientHello)**									
type	protocol version		length (2byte)		msg type	length (3byte)			client version		random	session id	cipher suite	compression	Extension
	major	minor							major	minor					
0x16	0x03	0x03	??	??	0x01	??	??	??	0x03	0x03	32 byte	可変	可変	可変	可変

Version

0x03,0x00 = SSLv3
0x03,0x01= TLSv1.0
0x03,0x02=TLSv1.1
0x03,0x03=TLSv1.2

```
ohtsu@ohtsu-mac:seccamp2016$ node tls_client.bot
TLSBot only supports TLS1.2(ECDHE-RSA-CHACHA20-POLY1305 with prim
Start with HelloRequest: 1603030004000000000
TLS Client> 1603030004000000000
<= HelloRequest
SECCAMP2016 53795: { ContentType: <Buffer 16>,
  ProtocolVersion: <Buffer 03 03>,
  Length: <Buffer 00 45>,
  Handshake:
   { HandshakeType: <Buffer 01>,
     Length: <Buffer 00 00 41>,
     ProtocolVersion: <Buffer 03 03>,
     Random: <Buffer f4 e1 10 5e 0d 1f d2 a9 b8 d2 6f df 6b b7 ac
     SessionID: <Buffer >,
     CipherSuites: [ <Buffer cc a8> ],
     CompressionMethods: <Buffer 00>,
     Extensions:
      [ { Type: <Buffer 00 0b>, Data: <Buffer 01 00> },
        { Type: <Buffer 00 0a>, Data: <Buffer 00 02 00 17> },
        { Type: <Buffer 00 0d>, Data: <Buffer 00 02 04 01> } ] },
  remaining_buffer: <Buffer > }
ClientHello => 160303004501000041030 3f4e1105e0d1fd2a9b8d26fdf6bb7
000200170 00d000400020401
TLS Client>
```

ServerHello

ClientHello

ServerHello

ClientHelloとServerHelloのやり取りで双方が利用するTLSバージョンや暗号化方式などを合意する。

(赤文字はハンドシェイク)

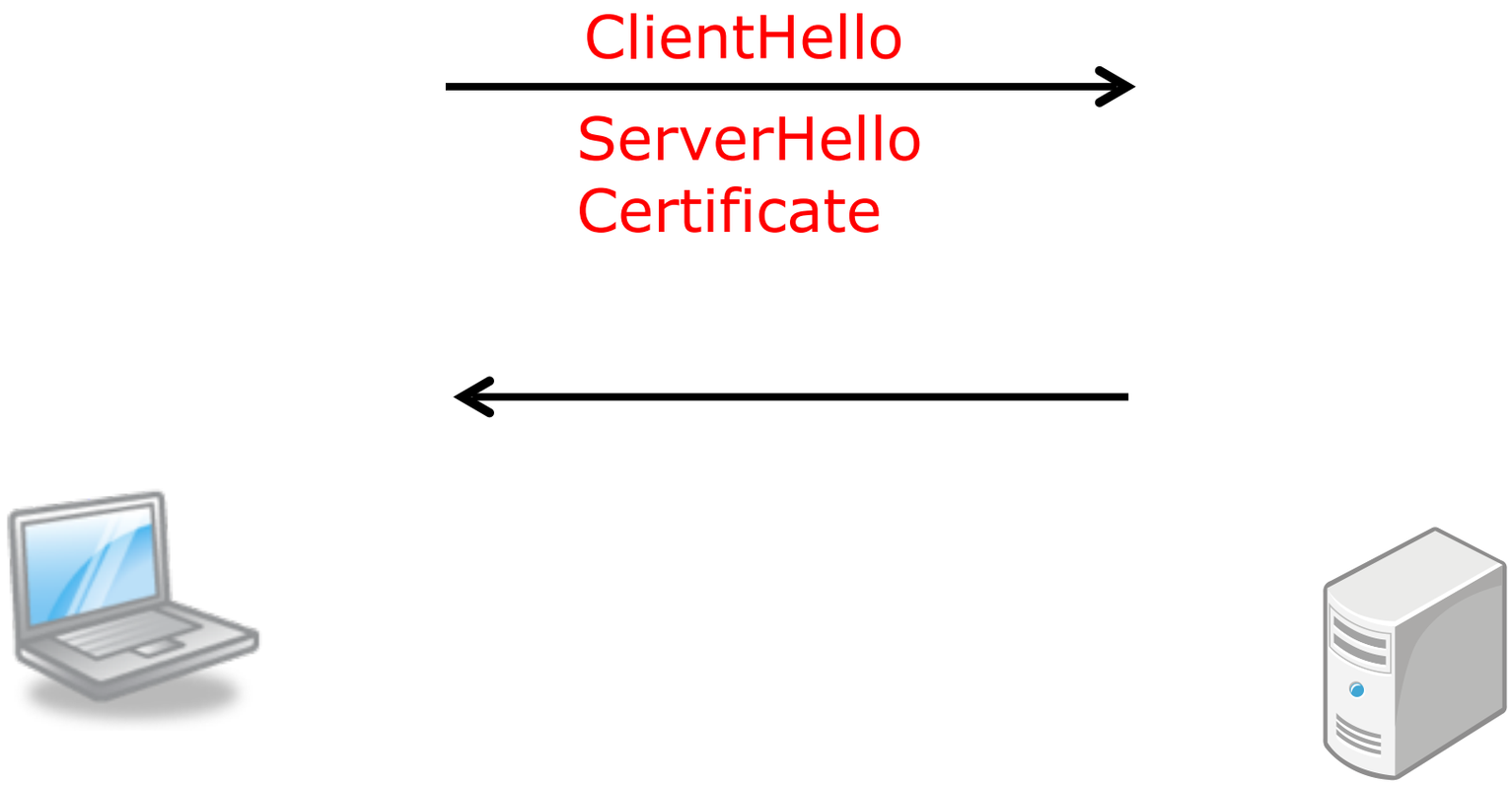
ServerHello

項目	要素	サイズ	先頭の長さ情報
server_version	uint8 major, uint8 minor	2	N/A
random	uint32 gmt_unix_time, opaque random_bytes[28]	4 + 28	N/A
session_id	opaque SessionID	<0..32>	1
cipher_suite	uint8 CipherSuite[2]	**2**	**N/A**
compression_method	null(0)	**1**	**N/A**
extensions	extension_type, extension_data<0..2^16-1>	<0..2^16-1>	2バイト分

Record Layer(5bytes)				**Handshake (ServerHello)**								
type	protocol version		length (2bytes)	msg type	length (3byte)	server version		random 32bytes	session id		cipher suite 2bytes	compression
	major	minor				major	minor					
0x16	**0x03**	**0x03**	**? + 4**	**0x01**	**?**	**0x03**	**0x03**	**?**	**長さ1byte**		**0x00,0x9c**	**長さ2bytes**

```
SECCAMP2016 53794: { ContentType: <Buffer 16>,
  ProtocolVersion: <Buffer 03 03>,
  Length: <Buffer 00 32>,
  Handshake:
   { HandshakeType: <Buffer 02>,
     Length: <Buffer 00 00 2e>,
     ProtocolVersion: <Buffer 03 03>,
     Random: <Buffer be f9 cc 0c f5 45 76 33 c8 00 50 d1 e9 dd ad e5 e3
     SessionID: <Buffer >,
     CipherSuite: <Buffer cc a8>,
     CompressionMethod: <Buffer 00>,
     Extensions: [ { Type: <Buffer 00 0b>, Data: <Buffer 01 00> } ] },
  remaining_buffer: <Buffer > }
ServerHello => 160303003202000002e0303bef9cc0cf5457633c80050d1e9ddade5e3
```

Certificate



(赤文字はハンドシェイク)

Certificate

複数の証明書データを送付

項目	要素	サイズ
certificate_list	ASN.1Cert<2^24-1>	<0..2^24-1>

最初は必ずサーバ証明書

2つ目以降は中間証明書など

```
SECCAMP2016 53794: { ContentType: <Buffer 16>,
  ProtocolVersion: <Buffer 03 03>,
  Length: <Buffer 02 fa>,
  Handshake:
   { HandshakeType: <Buffer 0b>,
     Length: <Buffer 00 02 f6>,
     Certificates: [ <Buffer 30 82 02 ec 30 82 01 d4 02 02 03
 06 13 02 4a 50 31 0e 30 0c 06 03 55 04 ... > ] } }
Certificate => 16030302fa0b0002f60002f30002f0308202ec308201d4
035504080c05546f6b796f3113301106035504070c0a436869796f64612d6
c0c544c532045786572636973653116301406035504030c0d4f6874737520
307e310b300906035504061302a50310e300c06035504080c05546f6b796
47920436d702032303136311530130603550400b0c0c544c532045786572
f70d01010105000381d003081890281810c70cde65036ecb4f613d86eb6
2585426674e7ea50016d181a46312e486d89e07c77edf65e15d11e78f9071
f6f61074910ee702030100013300d06092a864886f70d01010b050003820100
ea58b9279c82cbf859ed71b485326da17ec569647599ac2bd0b378fc615da
e629f4548b7bb29606ea811e3a3c8220c925115d4ff2bd99562447981b5df
237c626c2de2930fe4951731711735545bb219df7ee5a73448869a75d05b73
d93da3a7fd9ef410226d
```

Perfect Forward Secrecy(PFS)

- 前方秘匿性
    - セッション毎に一時的な鍵を使う。
    - ハンドシェイクを含む全暗号データを取得されているような状況でも、将来的な秘密鍵漏洩などのリスクに対応する。

TLS_ECDH**E**_RSA_WITH_AES_128_GCM_SHA256

Ephemeral:一時的な
鍵交換手法

DHE vs ECDHE

- DH: Diffe-Hellman 離散対数問題を利用した鍵交換

((g^x) mod P)^y mod P = ((g^y) mod P)^x mod P = g^(xy) mod P

素数P, ジェネレータ g, 公開鍵(赤字、青字）などの情報を交換。ECDHEより計算量が多い。

- ECDHE： 楕円関数上での離散対数演算を利用した鍵交換

楕円関数のパラメータ・基点を名前で規定(secp256等)、公開鍵(楕円曲線上の点)を交換。DHより鍵長・計算量が少なくてすむ。

ECDHEのハンドシェイク

使える楕円曲線名と楕円点書式を通知

楕円点の書式を合意

ServerHello拡張を追加

ClientHello拡張を追加

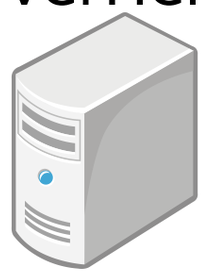
ClientHello
+ elliptic_curves
+ ec_point_formats

ServerHello
+ ec_point_formats

TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

Certificate
ServerKeyExchange
ServerHelloDone

Clientの公開鍵を送付



楕円曲線名とServerの公開鍵を署名付きで送付

ClientKeyExchange
ChangeCipherSpec
Finished

ChangeCipherSpec
Finished

Application Data

(赤文字が追加変更されるところ)

公開鍵は毎回ランダムに生成されます

ECDHE ClientHello拡張

クライアントがサポートしている楕円曲線のリストをサーバ側に通知。サーバはリストの中から適切な楕円曲線を選び ServerKeyExchange内で選択した楕円曲線を通知する

0	1	2	3	4	5	6	7
elliptic_curves(10)		リスト長		データ長		secp256r1 (23)	
0x00	0x0a	0x00	0x04	0x00	0x02	0x00	0x17

ECDHE Client/Server Hello拡張

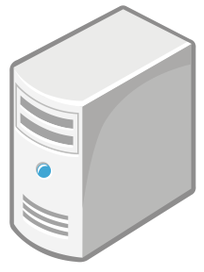
楕円暗号の公開鍵の書式

0	1	2	3	4	5
ec_point_formats(11)		リスト長		データ長	uncompressed(0)
0x00	0x0b	0x00	0x02	0x01	0x00

ServerKeyExchange

ClientHello

ServerHello
Certificate
ServerKeyExchange



(赤文字はハンドシェイク)

ECDHE ServerKeyExchange

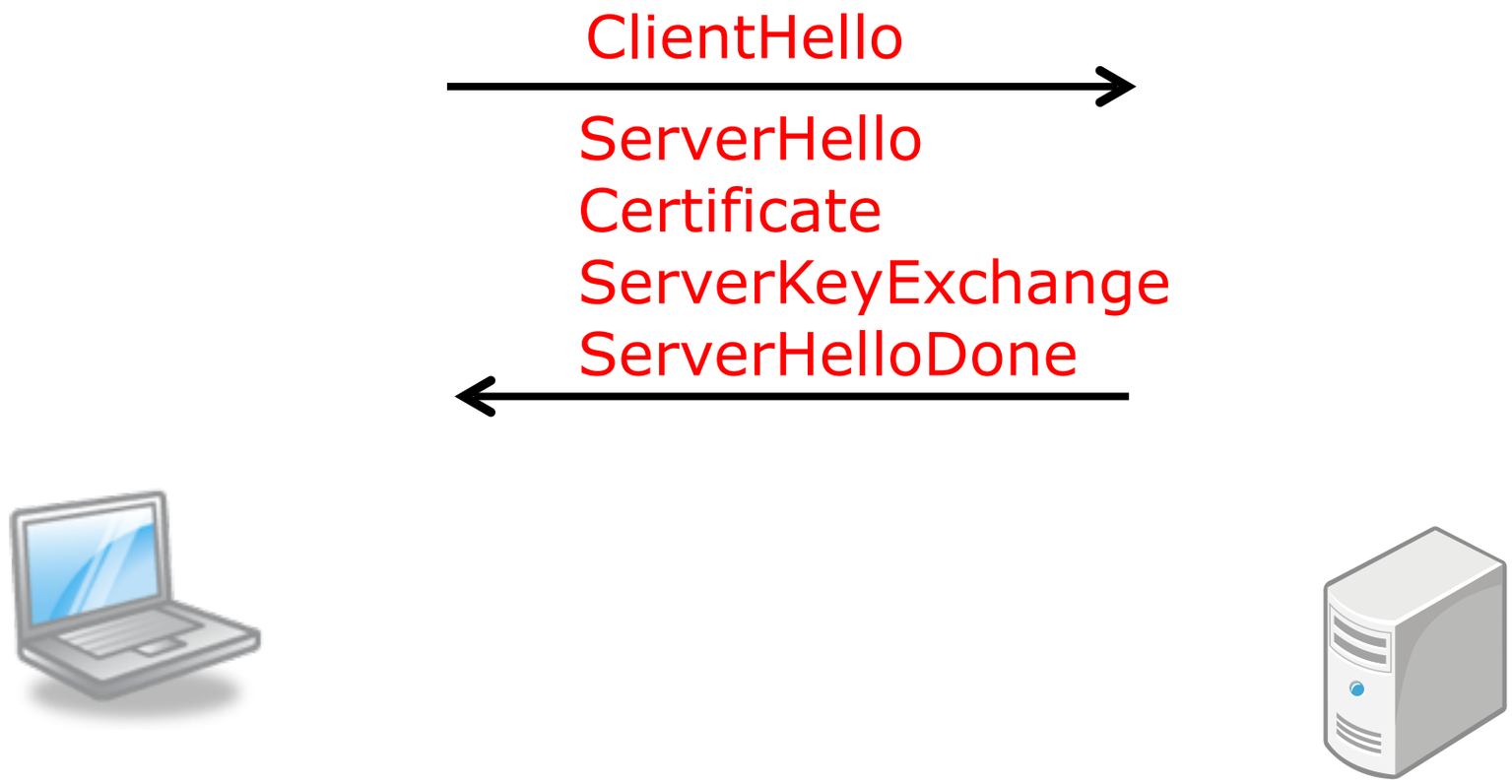
RSA秘密鍵でServerECDHParmsとRandomを署名

ServerECDHParams				Signature	
ECParameters		ECPoint		algorithm	signature
curve_type	named_curve	長さ	public key (Hello拡張指定の書式)	RSA-SHA256 (0x04,0x01)	
named_curve (3)	secp256r1 (23)				

signature = sign(algorithm, ClientHello.random + ServerHello.random + ServerECDHParams);

```
SECCAMP2016 53794: { ContentType: <Buffer 16>,
  ProtocolVersion: <Buffer 03 03>,
  Length: <Buffer 00 cd>,
  Handshake:
   { HandshakeType: <Buffer 0c>,
     Length: <Buffer 00 00 c9>,
     ECCurveType: <Buffer 03>,
     ECNamedCurve: <Buffer 00 17>,
     ECPublic: <Buffer 04 f5 9e 63 7f 7f 08 3b c3 f6 4f ce 93 1f 0e
 23 7d 80 9f 67 71 e9 85 4a 6d a0 ... >,
     ECSignatureHashAlgorithm: <Buffer 04 01>,
     ECSignature: <Buffer 28 09 19 cf 69 61 fd c2 99 b3 51 24 07 3c
 ec fa 60 bf 3e ae ee 07 e4 cd f2 79 ... > } }
ServerKeyExchange => 16030300cd0c0000c903001741 04f59e637f7f083bc3f6
854a6da0f902120a98456d0ef1e04e801b4dfc04010080280919cf6961fdc299b35
4cdf2796ed983d750e845dac39f3d0f3c9b27bfa24060f71a2f66142647 09e4f6f0
a8387818d8070d1ca5ff0ec0
```

ServerHelloDone



(赤文字はハンドシェイク)

ServerHelloDone

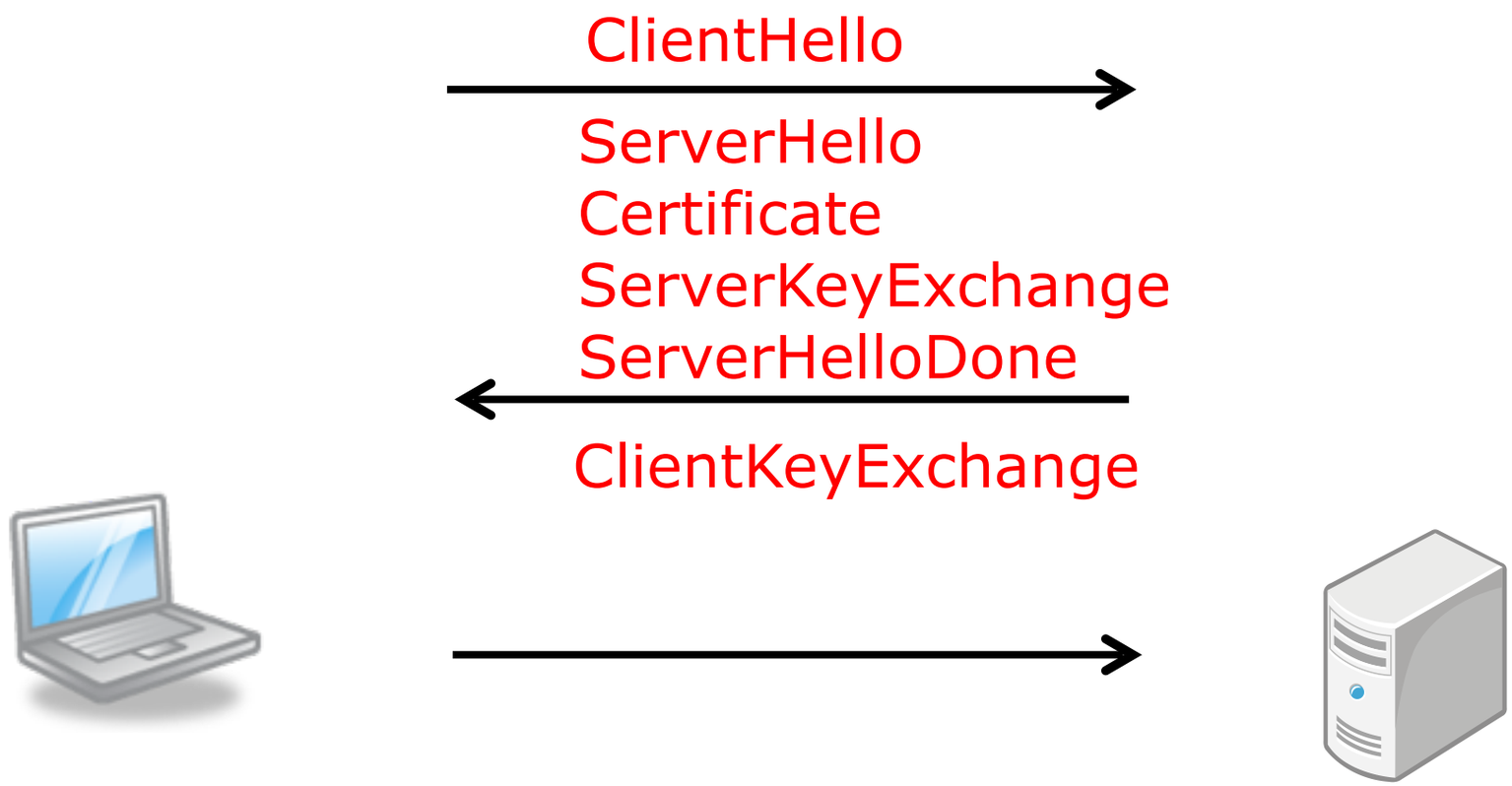
ここでServerHelloから続く一連のハンドシェイクの前半が終了したことを告げる合図

handshake type	handshake長		
0x0e	0x00	0x00	0x00

ServerHelloの終了の合図
ハンドシェイクヘッダのみ

```
SECCAMP2016 53794: { ContentType: <Buffer 16>,
  ProtocolVersion: <Buffer 03 03>,
  Length: <Buffer 00 04>,
  Handshake: { HandshakeType: <Buffer 0e>, Length: <Buffer 00 00 00> } }
ServerHelloDone => 16030300040e000000
```

TLSハンドシェイク(full handshake)



(赤文字はハンドシェイク)

ECDHE ClientKeyExchange

ClientECDHParams	
ECPoint	
長さ	public key (Hello拡張指定の書式)

ClientKeyExchangeは署名の必要はない

```
SECCAMP2016 53795: { ContentType: <Buffer 16>,
  ProtocolVersion: <Buffer 03 03>,
  Length: <Buffer 00 46>,
  Handshake:
   { HandshakeType: <Buffer 10>,
     Length: <Buffer 00 00 42>,
     ECPublic: <Buffer 04 6f fb 1b e6 2c 2b 65 f1 c
 cd 20 67 ab 52 b2 5c 01 95 66 db ... > } }
ClientKeyExchange => 160303004610000042410 46ffb1be6
db08ddadfdbaf7e32bb2e67d819f1b44
```

質問： ECDHE公開鍵の守られ方の違い

- ServerKeyExchange: 公開鍵を署名
- ClientKeyExchange: やりたい放題

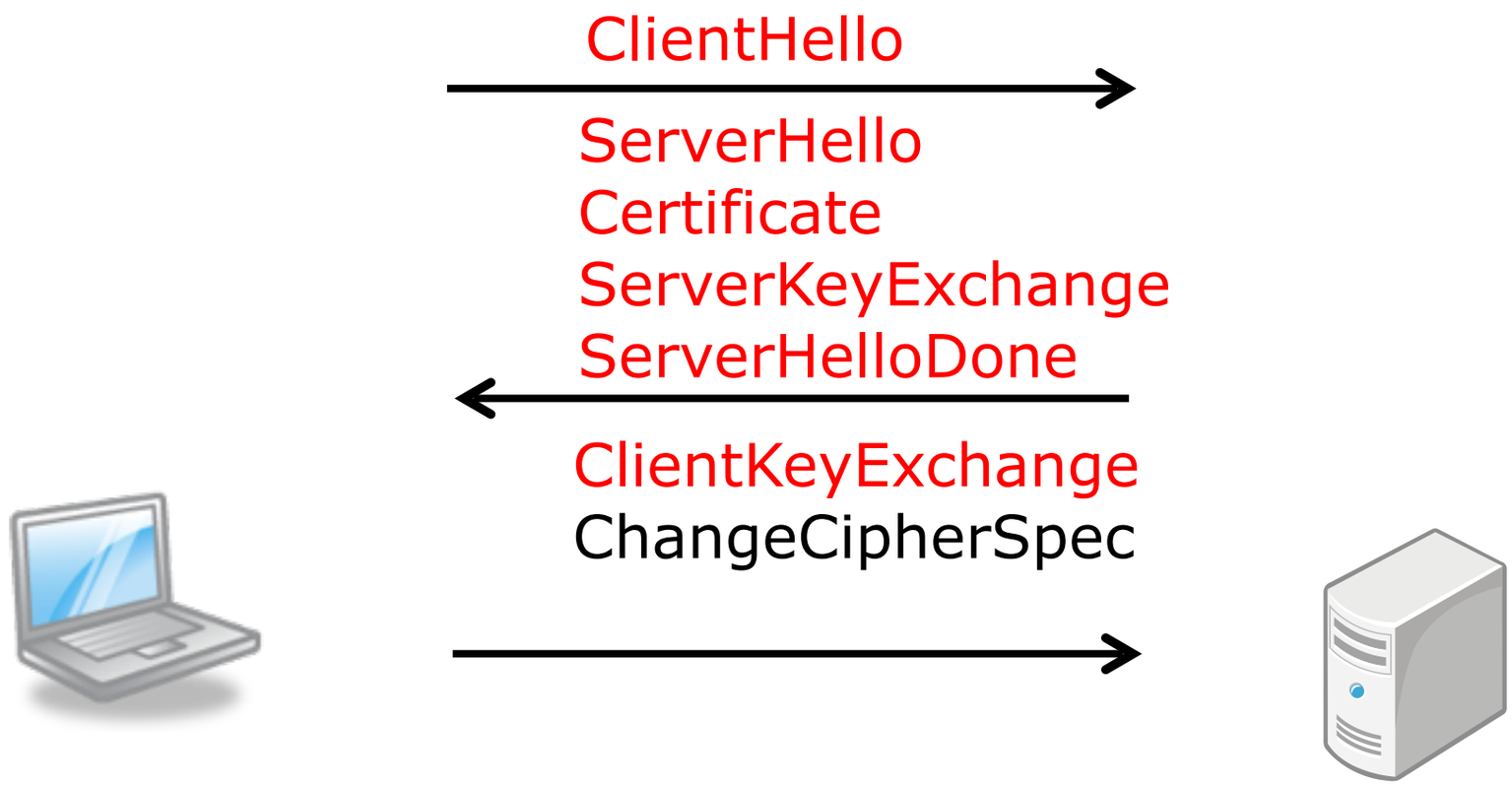
どうしてでしょう？

PreMasterSecret/MasterSecret

- TLSで利用するIV(初期ベクトル)、共有鍵、MAC鍵のデータ元
- MasterSecretは48バイト長。PreMasterSecretの長さは鍵交換方式に依存する。
- MasterSecretは、PreMasterSecret、ClientRandom、ServerRandom、固定ラベルから生成する。
- Clinet/ServerRandomは全て丸見え。PreMasterSecretは、必ず死守して守らないといけない。これが漏えいするとTLSの安全性は全ておじゃん。

```
TLS Client> .info
{ client_write_key: 'fc487a3966db0ddbe2334663b841af96e2c569bc37cc5e89cef0d191f371da88',
  server_write_key: '42432de41c40c0942fc3a08c37c55991726df9e68beada492b03021ea979af3b',
  client_write_iv: 'da83ba39ee4db530f22a818b',
  server_write_iv: '9adc331fffd65c80e3fa2f4a' }
TLS Client>
```

ChangeCipherSpec
Client->Server



(赤文字はハンドシェイク)

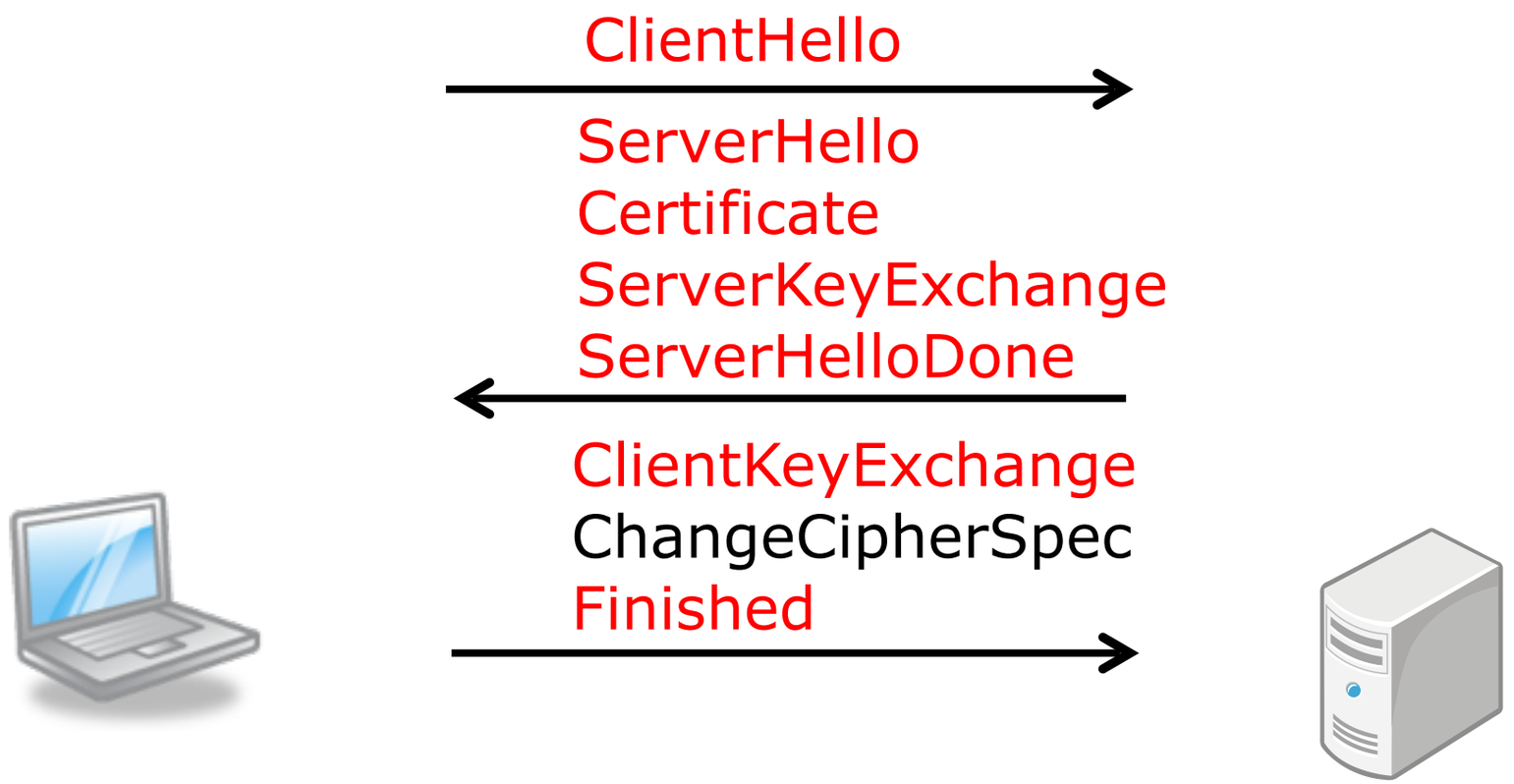
ChangeCipherSpec

送信元が暗号開始を宣言。これを送信した後は暗号通信を行う。

Record Layer					**ChangeCipherSpec**
ContentType	Version		length (2byte)		
	major	minor			
0x14	0x03	0x03	0x00	0x01	0x01

```
SECCAMP2016 53795: { ContentType: <Buffer 14>,
  ProtocolVersion: <Buffer 03 03>,
  Length: <Buffer 00 01>,
  ChangeCipherSpecMessage: <Buffer 01> }
ChangeCipherSpec => 140303000101
        (Encrypt Write Start)
```

TLSハンドシェイク(full handshake)



(赤文字はハンドシェイク)

Finished

```
struct {
    opaque verify_data[verify_data_length];
} Finished;
```

12バイト固定

これまでのハンドシェイクデータ（ただし自分は除く）のハッシュを計算
TLS1.2では SHA256を使う

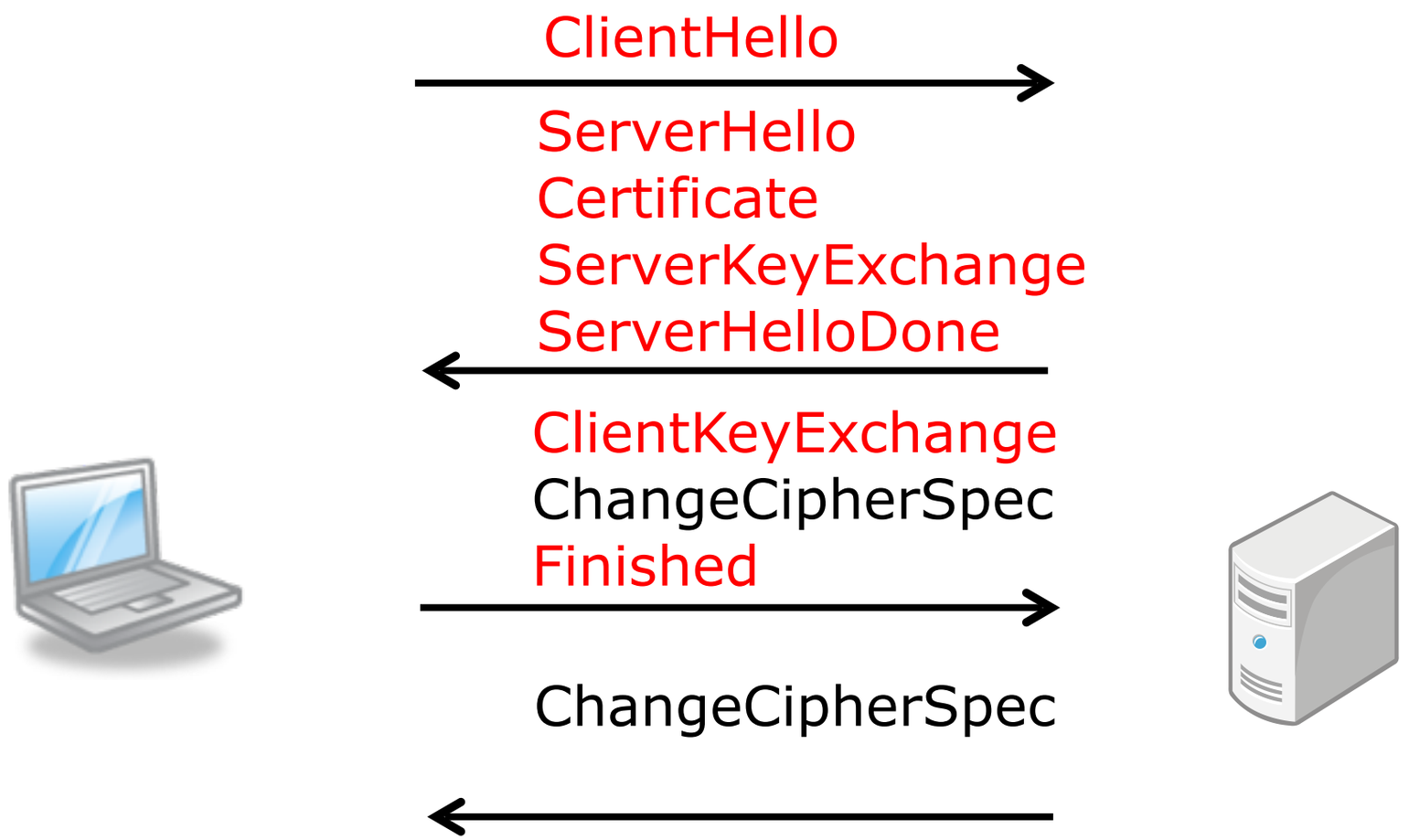
verify_data = PRF(master_secret, finished_label,
Hash(handshake_messages))[0..11];

finished_label:　クライアントは、"client finished"、サーバは"server finished"

Finishedを受信すると、これまで送受信したハンドシェイクデータから計算した値と比較。
ハンドシェイクデータが改ざんされてないことを確認する。

```
SECCAMP2016 53795: { ContentType: <Buffer 16>,
  ProtocolVersion: <Buffer 03 03>,
  Length: <Buffer 00 10>,
  Handshake:
   { HandshakeType: <Buffer 14>,
     Length: <Buffer 00 00 0c>,
     VerifyData: <Buffer 79 64 98 21 8d b2 5f 2a 05 79 61 1d> } }
ClientFinished => 1603030020e278b57b669f87e285e4302625aaa4eb4f7f83
```

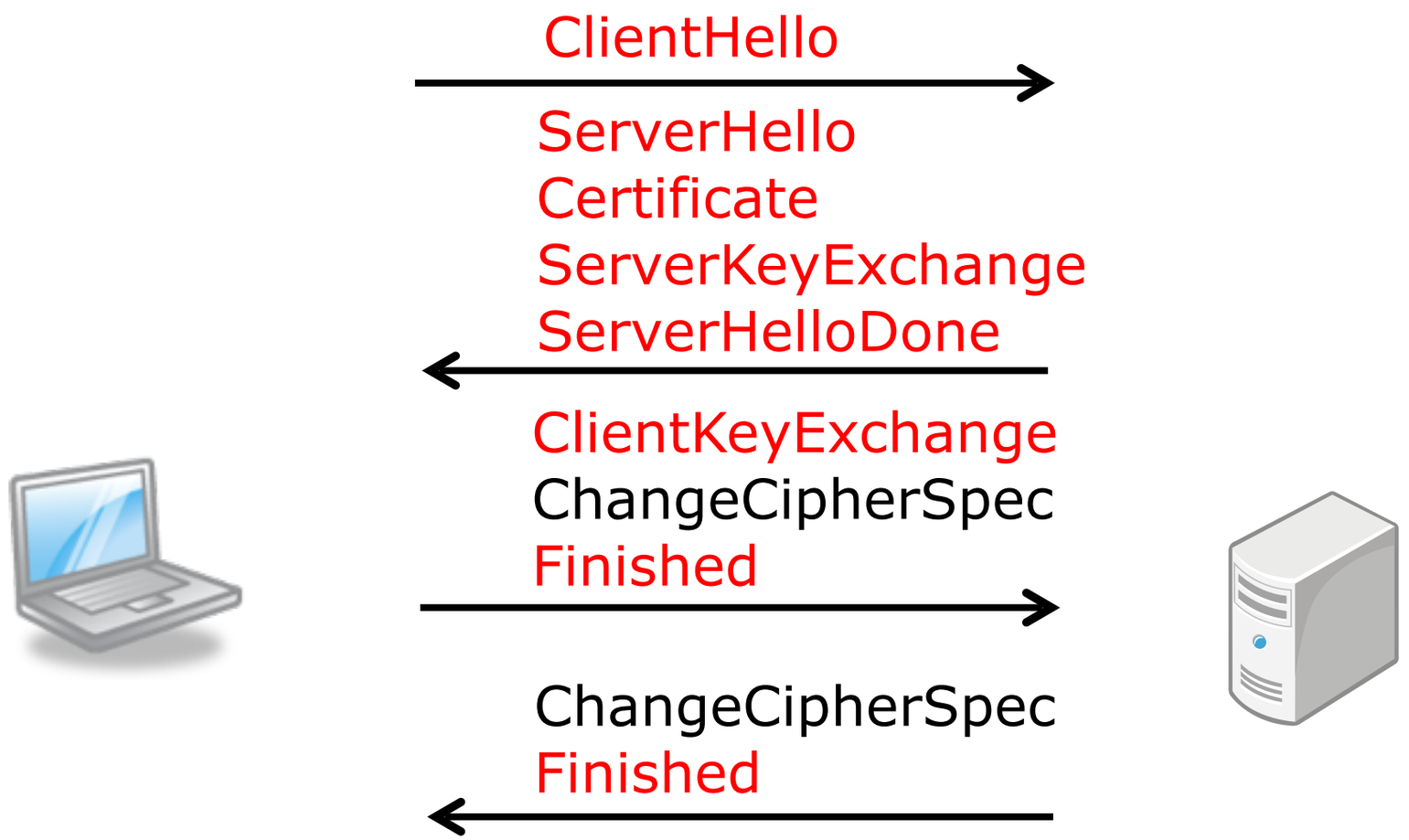
ChangeCipherSpec
Server -> Client



(赤文字はハンドシェイク)

```
<= ChangeCipherSpec
        (Encrypt Read Start)
TLS Server> 1603030020e278b57b669f87e285e430262
<= Finished
        (Handshake Verified and Completed)
SECCAMP2016 53794: { ContentType: <Buffer 14>,
  ProtocolVersion: <Buffer 03 03>,
  Length: <Buffer 00 01>,
  ChangeCipherSpecMessage: <Buffer 01> }
ChangeCipherSpec => 140303000101
        (Encrypt Write Start)
```

ServerFinished



(赤文字はハンドシェイク)

```
SECCAMP2016 53794: { ContentType: <Buffer 16>,
  ProtocolVersion: <Buffer 03 03>,
  Length: <Buffer 00 10>,
  Handshake:
   { HandshakeType: <Buffer 14>,
     Length: <Buffer 00 00 0c>,
     VerifyData: <Buffer 3f af 5d 68 be f6 7b 5d 99 24 05 32> } }
ServerFinished => 1603030020caf364cfb56bea653ec983b9ec05ab24bf8856
```

```
TLS Client> 140303000101
<= ChangeCipherSpec
        (Encrypt Read Start)
TLS Client> 1603030020caf364cfb56bea653ec983
<= Finished
        (Handshake Verified and Completed)
```

Application Data

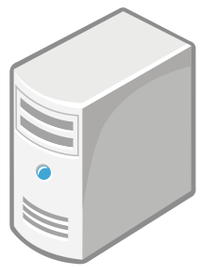
ClientHello

ServerHello
Certificate
ServerKeyExchange
ServerHelloDone

ClientKeyExchange
ChangeCipherSpec
Finished

ChangeCipherSpec
Finished

Application Data



(赤文字はハンドシェイク)

```
TLS Client> 'hoge'
ApplicationData => 170303001460 63fc34990515a9b9dc708bfaab232c9206f711
TLS Client>
```

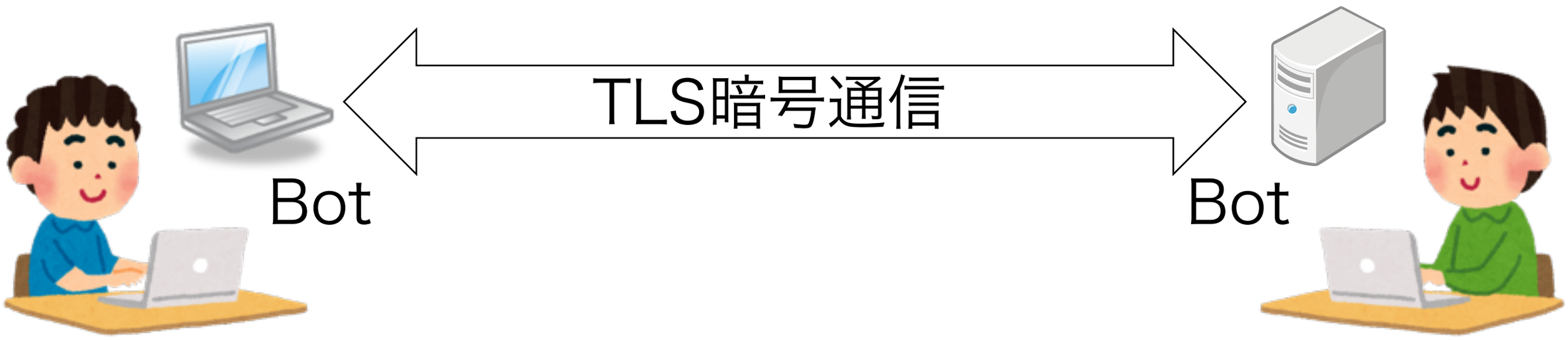
```
TLS Server> 170303001460 63fc34990515a9b9dc708bfaab232c9206f711
<= ApplicationData hoge
TLS Server>
```

演習

TLS Botを使った1対1 TLS

- 二人一組になってBotを使ったTLS通信を行います。 Client役、Server役を決めて下さい。

- サイボウズでデータをやり取りします。相互で暗号文の復号化ができることを確認します。



TLS Server> 17030300146063fc34990515a9b9dc708bfaab232c9206f711

<= ApplicationData hoge

演習

リアルMan-In-The-Middle

- ３人一組になってBotを使ったTLS通信を行います。 Client役、悪人役、Server役を決めて下さい。

- 悪人役を介してTLSハンドシェイクを行います。まずは悪人はそのまま右から左に受け流します。

- 次に悪人役でServer/ClientのBotをたててなりすまし通信をしましょう。

2016年６月に仕様化完了(RFC7905)した
TLSの新しい暗号方式

ChaCha20-Poly1305

私は暗号専門家ではないので安全性のお話はスキップします。

ChaCha20-Poly1305

- ChaCha20: D. J. Bernstein(djb)氏が考案した暗号方式(最初にSalsa20を発表、ChaCha20に改良)
- Poly1305: djb氏が考案したMAC方式(AESと組み合わせたAES-Poly1305で発表)

基本両者は独立したもの。GoogleのAdam Langley氏がChaCha20-Poly1305として2013/09にドラフト仕様を公開

ChaCha20-Poly1305 これまでの歩み

2005/03	Poly1305論文発表(最終版)
2008/01	ChaCha20論文発表(最終版)
2008/04	Salsa20がeStreamのFinalistに選定
2010/12	ChaCha20を使ったBLAKEがSHA-3の最終候補に選定
2013/09	**draft-agl-tls-chacha20poly1305-00 公開**
2013/11	ChromeがChaCha20-Poly1305を実装。Googleサービスで利用開始
	OpenSSHがChaCha20-Poly1305を実装
2014/02	TLS WGからCFRGへChaCha20-Poly1305の仕様検討を進めることを要請
2014/04	LibreSSLがfork。ChaCha20-Poly1305を実装
2014/06	BoringSSLがfork。 ChaCha20-Poly1305を実装
2015/02	CloudFlareがChaCha20-Poy1305の利用開始。OpenSSL用パッチ公開
2015/05	**RFC7539 (ChaCha20-Poly1305仕様)が公開**
2015/12	OpenSSL-1.1.0(alpha)がChaCha20-Poly1305を実装
2016/03	FirefoxがChaCha20-Poy1305を実装
2016/06	**RFC7905 ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)**

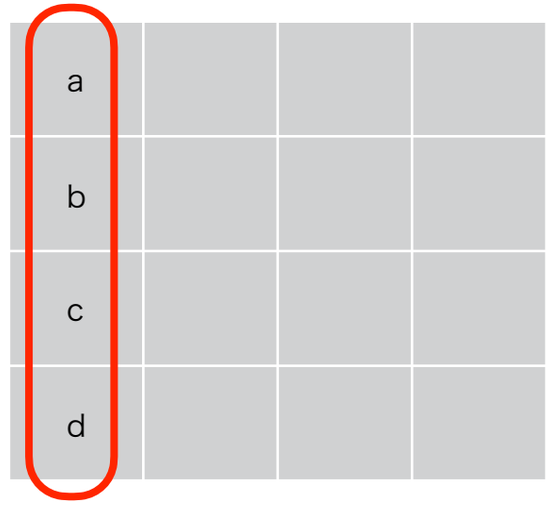
2013/06
Snowden事件

AESとChaCha20の比較

	AES	ChaCha20
方式	ブロック(128bits) (*1)	ストリーム
入力	鍵長: 128, 192, 256 bits	鍵長: 256bits
	Nonce: なし 初期カウンター: なし	Nonce: 96bits (djb論文では64bits) 初期カウンター: 32bits
標準	NIST FIPS-197	RFC7539(*2) (Salsa20はestreamソフトウェアPh3選定)
性能特性	AES-NIなど専用ハードウェアによる高速処理が可能	事前計算やSBOXが必要なく、タイミング攻撃が発生しにくい。SIMDを使った高速なソフトウェア処理が可能
注意事項	キャッシュタイミングなどサイドチャネル攻撃に対応した実装であること	Nonceを再利用しないこと

(*1 カウンターモードと組み合わせてストリーム暗号として利用が可能)
(*2 DJBの論文 http://cr.yp.to/chacha/chacha-20080128.pdf がアルゴリズム規定の参照先)

ChaCha 1/4ラウンド演算



QuarterRound(a, b, c, d)

1.　a += b,　d ^= a,　d <<<= 16

2.　c += d,　b ^= c,　b <<<= 12

3.　a += b,　d ^= a,　d <<<=　8

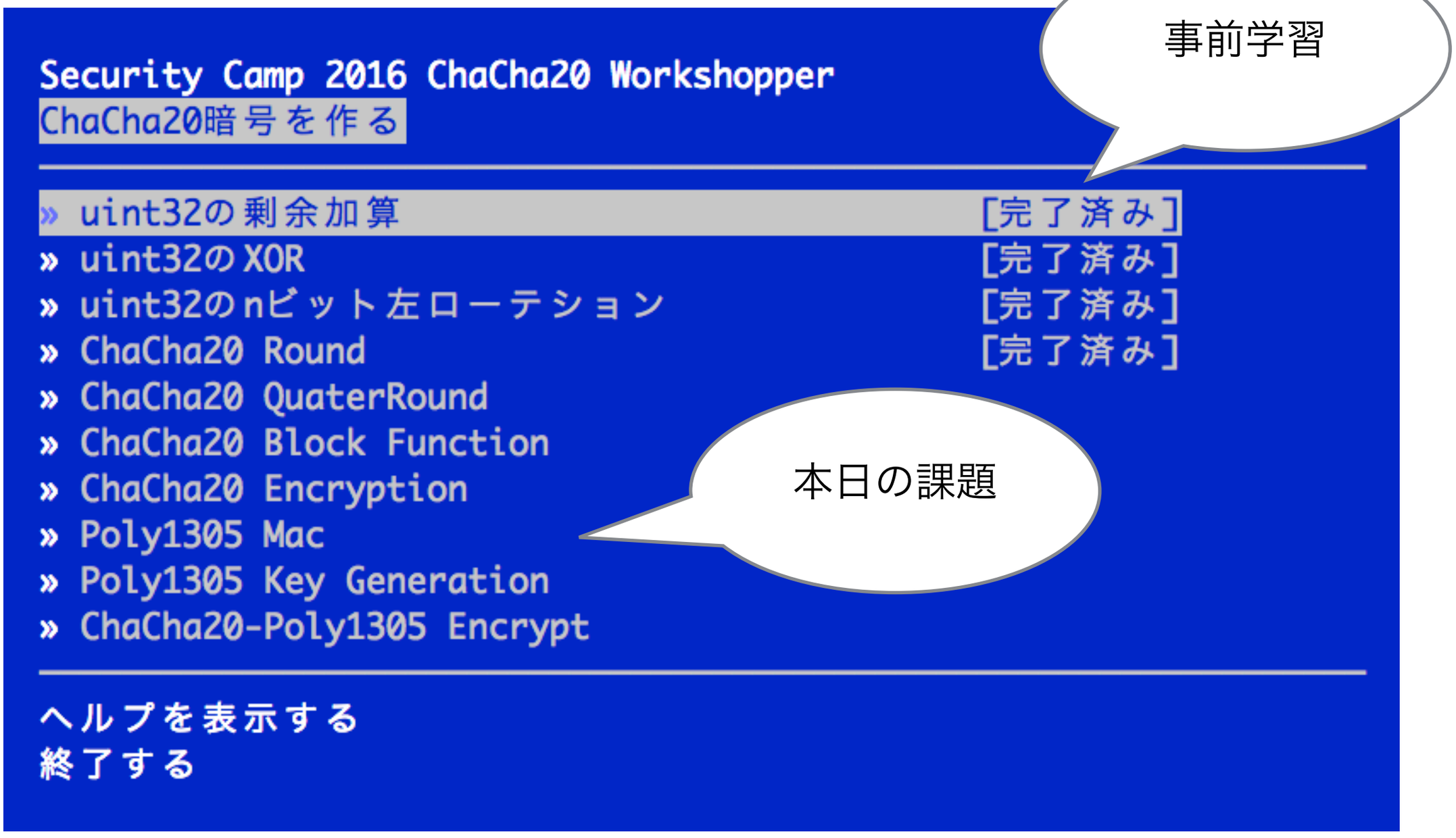
4.　c += d,　b ^= c,　b <<<=　7

乗算がなく固定長演算
Constant Time

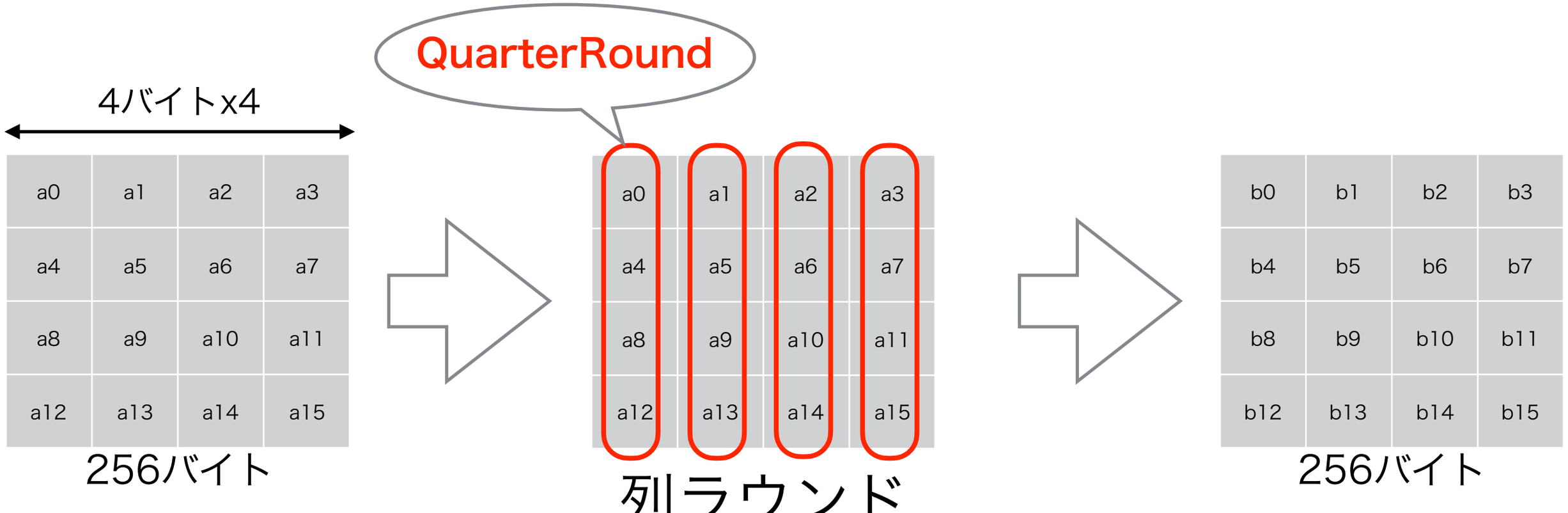
a, b, c, d に対して、全て2回
演算が行われている。

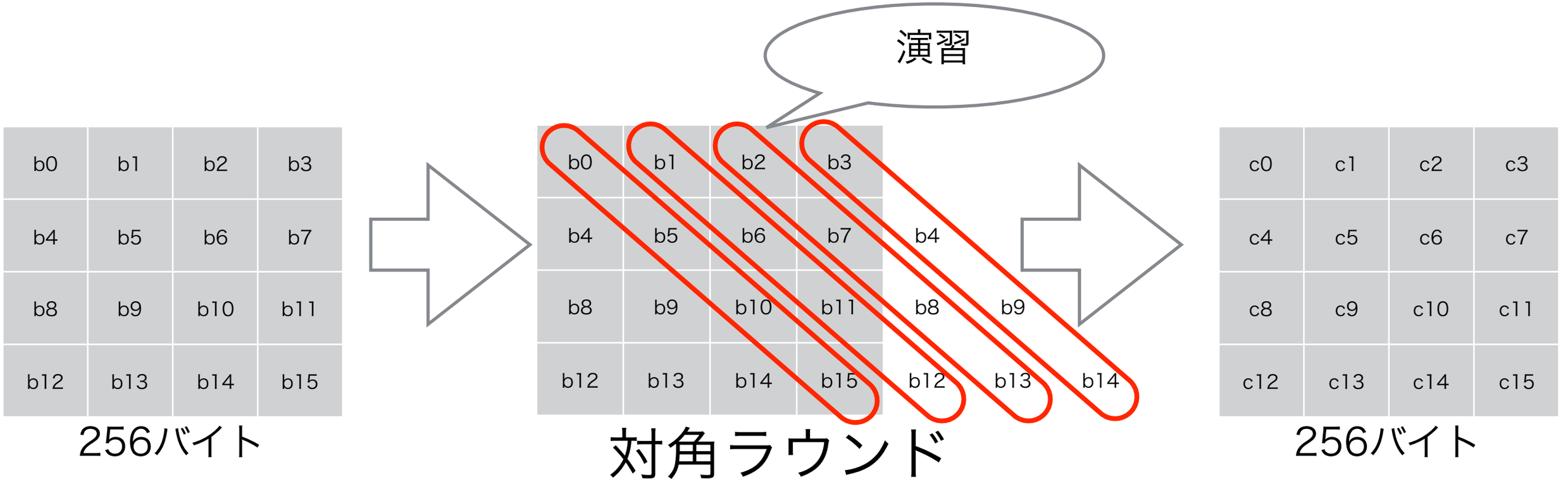
a, b, c ,d は 32-bit unsigned int
x + y は (x+y) mod 2^32,　^ は XOR,　<<< n はnビット左ローテション

課題・演習



sudo npm -g install seccamp2016-chacha20-workshopper





2種類のChaChaラウンド

初期ChaCha State

65 78 70 61	6e 64 20 33	32 2d 62 79	74 65 20 6b
key0	key1	key2	key3
key4	key5	key6	key7
counter	nonce0	nonce1	nonce2

定数値(実は以下の文字列)
expand 32-byte k

鍵(32バイト長)

Nonce(12バイト長)

1から始まるカウンター(4バイト長)

20ラウンド

(列ラウンド+対角ラウンド)x10回

ChaCha20 State

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15

ChaCha20 Stream State

ChaCha20 StateのEndianに注意

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k[0]	k[1]	k[2]	k[3]	k[4]	k[5]	k[6]	k[7]	k[8]	k[9]	k[10]	k[11]	k[12]	k[13]	k[14]	k[15]

4バイト毎に区切ったLittle Endian

65 78 70 61	6e 64 20 33	32 2d 62 79	74 65 20 6b
k[3]k[2]k[1]k[0]	k[7]k[6]k[5]k[4]	k[11]k[10]k[9]k[8]	k[15]k[14]k[13]k[12]
key4	key5	key6	key7
counter	nonce0	nonce1	nonce2

このようなデータ処理をする時は注意。見かけBig Endian。

ChaCha20 Block Function

初期ChaCha state

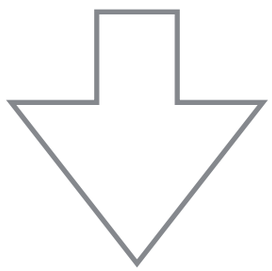
65 78 70 61	6e 64 20 33	32 2d 62 79	74 65 20 6b
key0	key1	key2	key3
key4	key5	key6	key7
counter	nonce0	nonce1	nonce2

+

(剰余和)

20 Round ChaCha state

s0	s1	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
s12	s13	s14	s15



Final ChaCha20 State

演習

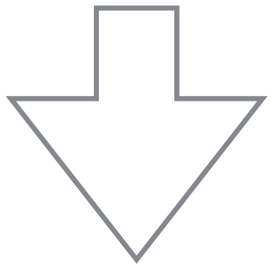
ChaCha20 QuaterRound

ChaCha20 BlockFunction

ChaCha20 KeyStream

03 02 01 00	07 06 05 04	0b 0a 09 08	0f 0e 0d 0c
13 12 11 10	17 16 15 14	1b 1a 19 18	1f 1e 1d 1c
23 22 21 20	27 26 25 24	2b 2a 29 28	2f 2e 2d 2c
33 32 31 30	37 36 35 34	3b 3a 39 38	3 f3e 3d 3c

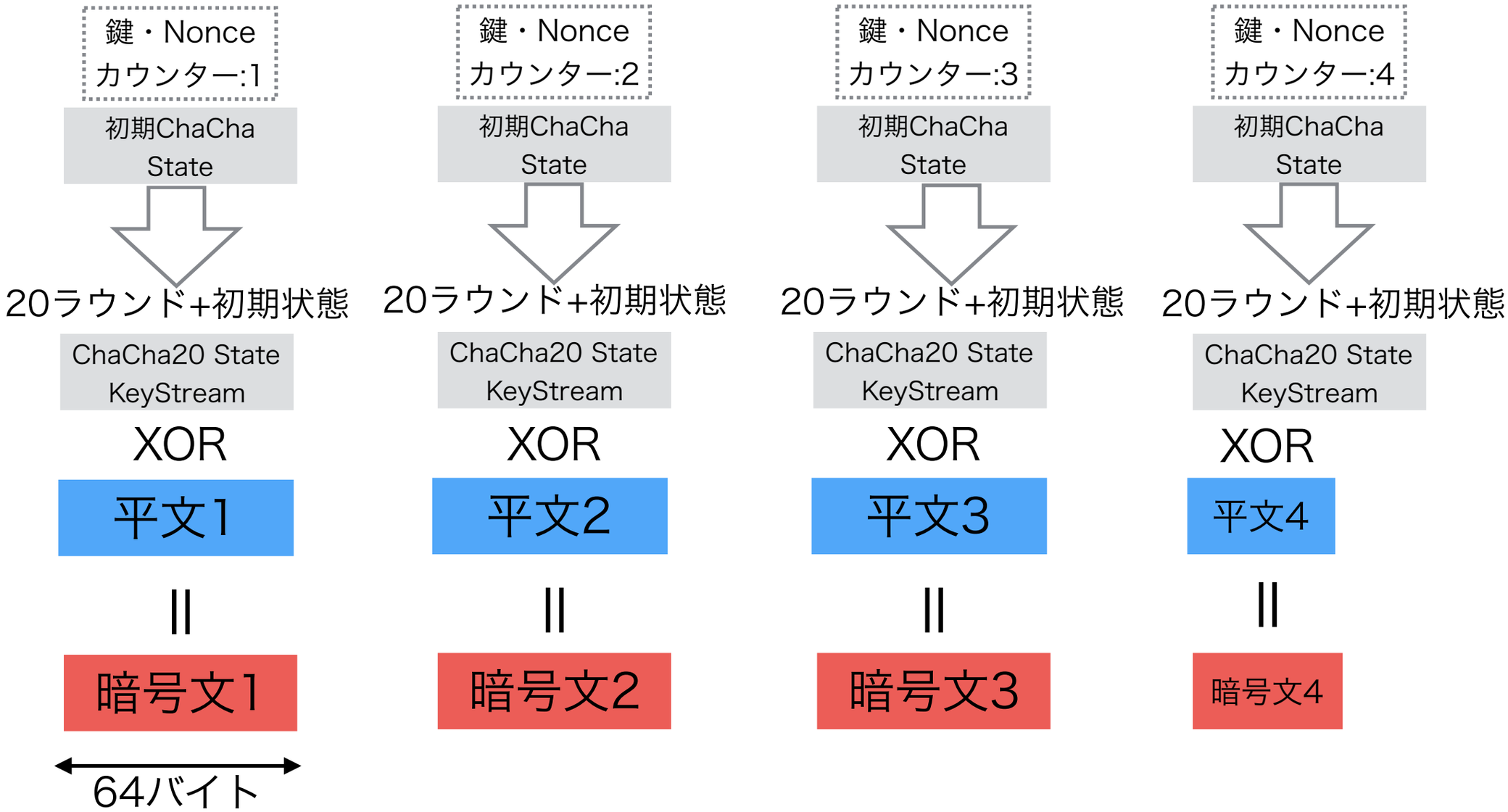
４バイト単位の各要素をLittle Endianで並び替え



000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f

KeyStreamと平文のXORを取って暗号文を生成する。

ChaCha20 平文の暗号化



復号化もKeyStreamと暗号文をXORするだけなので手順はほぼ同一

演習

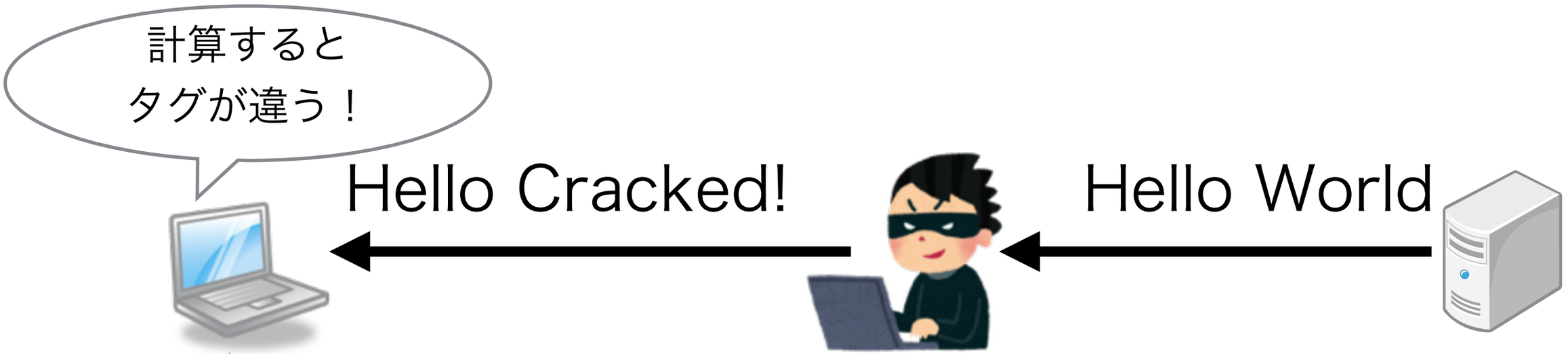
ChaCha20 KeyStream

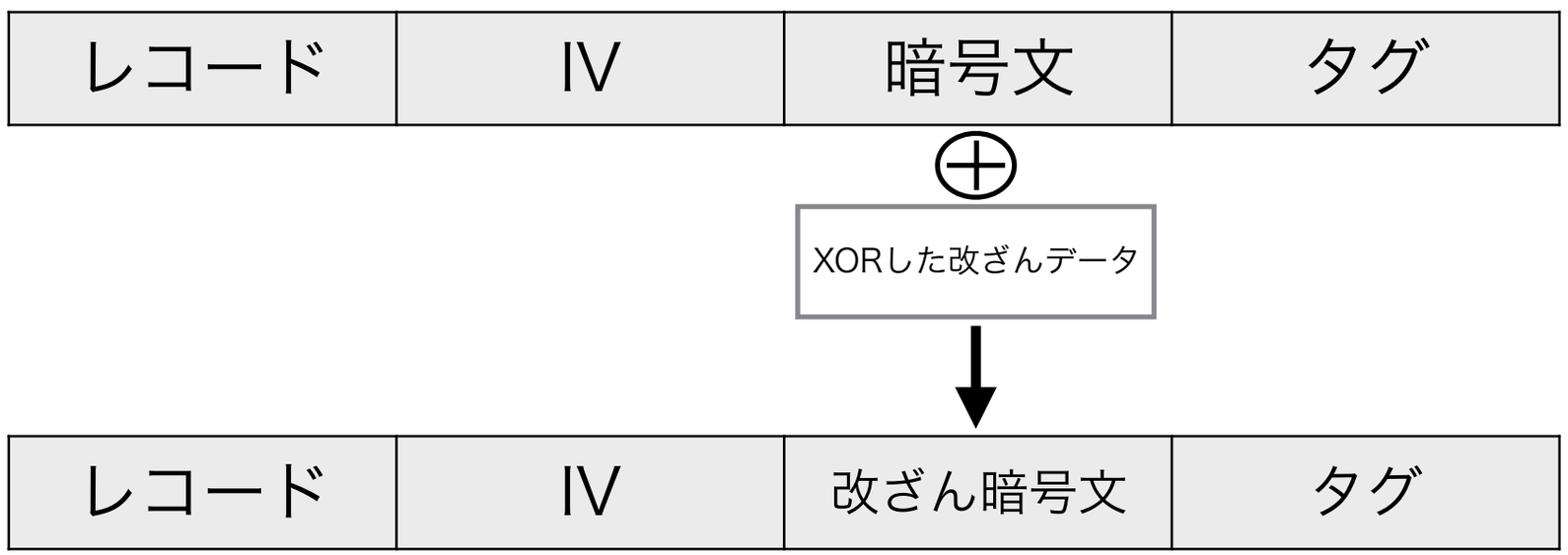
ChaCha20 Encryption

Poly1305

こちらは少々難しいので説明が主です

なぜ message authenticator が必要か？





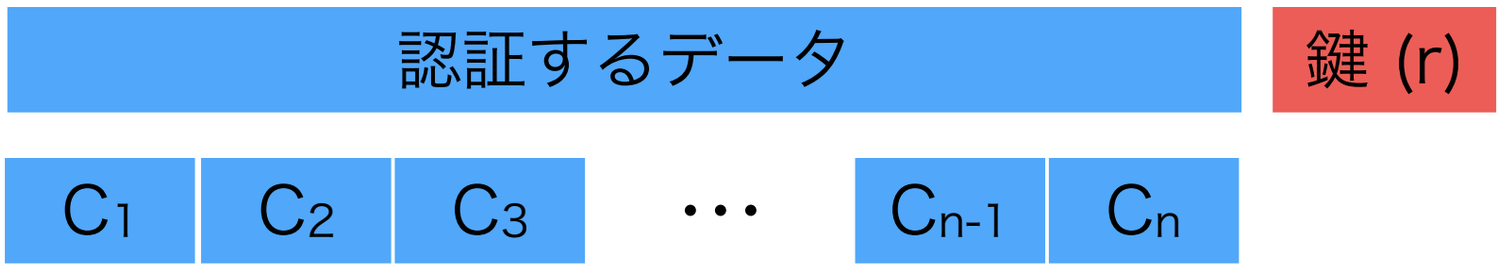
タグの再計算には秘密鍵が必要

GHASHとPoly1305

	GHASH	Poly1305
計算方式	Wegman-Carter Construction	
	binary field ($x^{128}+x^7+x^2+x+1$)	prime field ($2^{130}-5$)
鍵長	128bits (AESと組み合わせた時)	256bits
MAC長	128bits(利用目的に応じて切り詰める)	128bits
標準	NIST SP 800-38D(AES-GCM)	RFC7539(*)
性能特性	PCLMULQDQIなど特定計算用ハードウェアによる高速処理が可能	事前計算テーブルが必要なく、SIMDを使った高速なソフトウェア処理が可能
注意事項	· 鍵、IV(Nonce)を再利用しないこと · MAC長は64bits以上を利用すること	· 鍵、IV(Nonce)を再利用しないこと · タイミング攻撃に対応した実装であること

(* AESと組み合わせた DJBの論文 http://cr.yp.to/mac/poly1305-20050329.pdf がアルゴリズム規定の参照先)

Polynomial evaluation



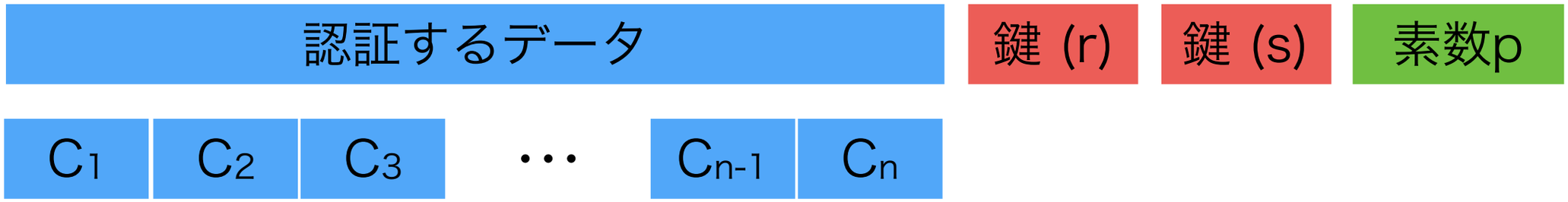
分解したメッセージを係数
とした多項式の値で評価

$$f(r)=C_1r^n+C_2r^{n-1}+C_3r^{n-2}+\cdots+C_{n-1}r^2+C_nr$$

$$((\cdots(C_1r+C_2)r+C_3)r+\cdots+C_{n-1})r+C_n)r$$

ホーナー法を使って乗算
演算を減らして計算

Wegman-Carter Construction for Poly1305

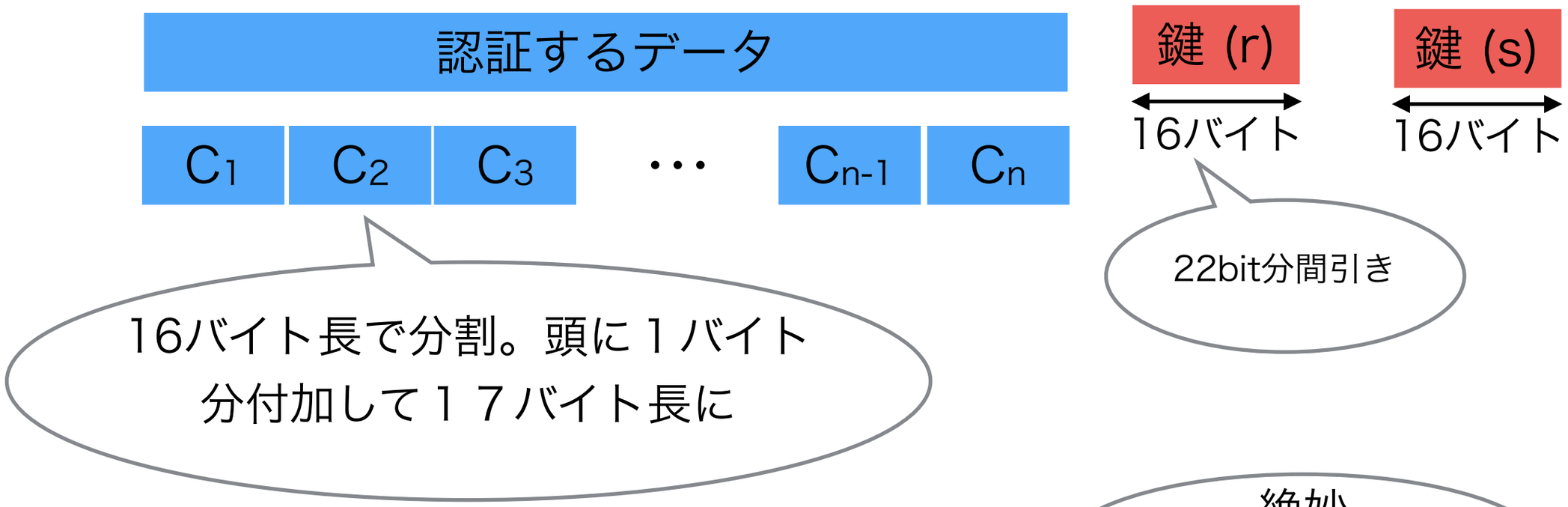


ユニバーサルハッシュ + OneTime鍵

$$C_1r^n+C_2r^{n-1}+C_3r^{n-2}+\cdots+C_{n-1}r^2+C_nr \bmod p + s$$

- 数学的に強度が証明できている
- SHA-1/2などのHMACより高速

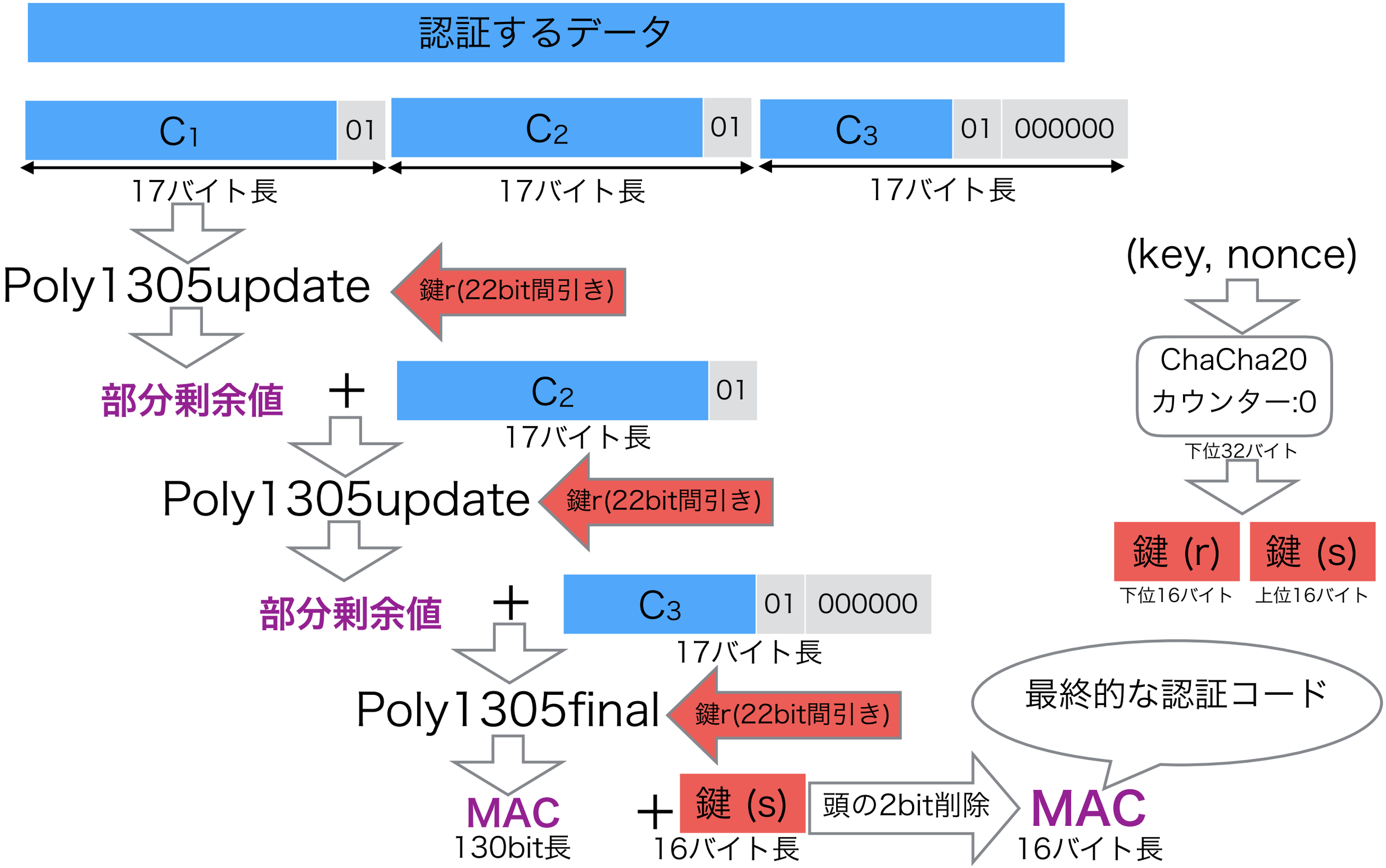
Poly1305



絶妙
なサイズの素数

$$(C_1r^n+C_2r^{n-1}+C_3r^{l-2}+\cdots+C_{n-1}r^2+C_nr \bmod 2^{130}-5 +s) \bmod 2^{128}$$

最終的に16バイト長
に切り詰める



Poly1305によるMACデータの生成

演習

Poly1305 MAC

さすがに短時間で実装してもらうのは辛いのでライブラリを使ってもらいます。

AEAD

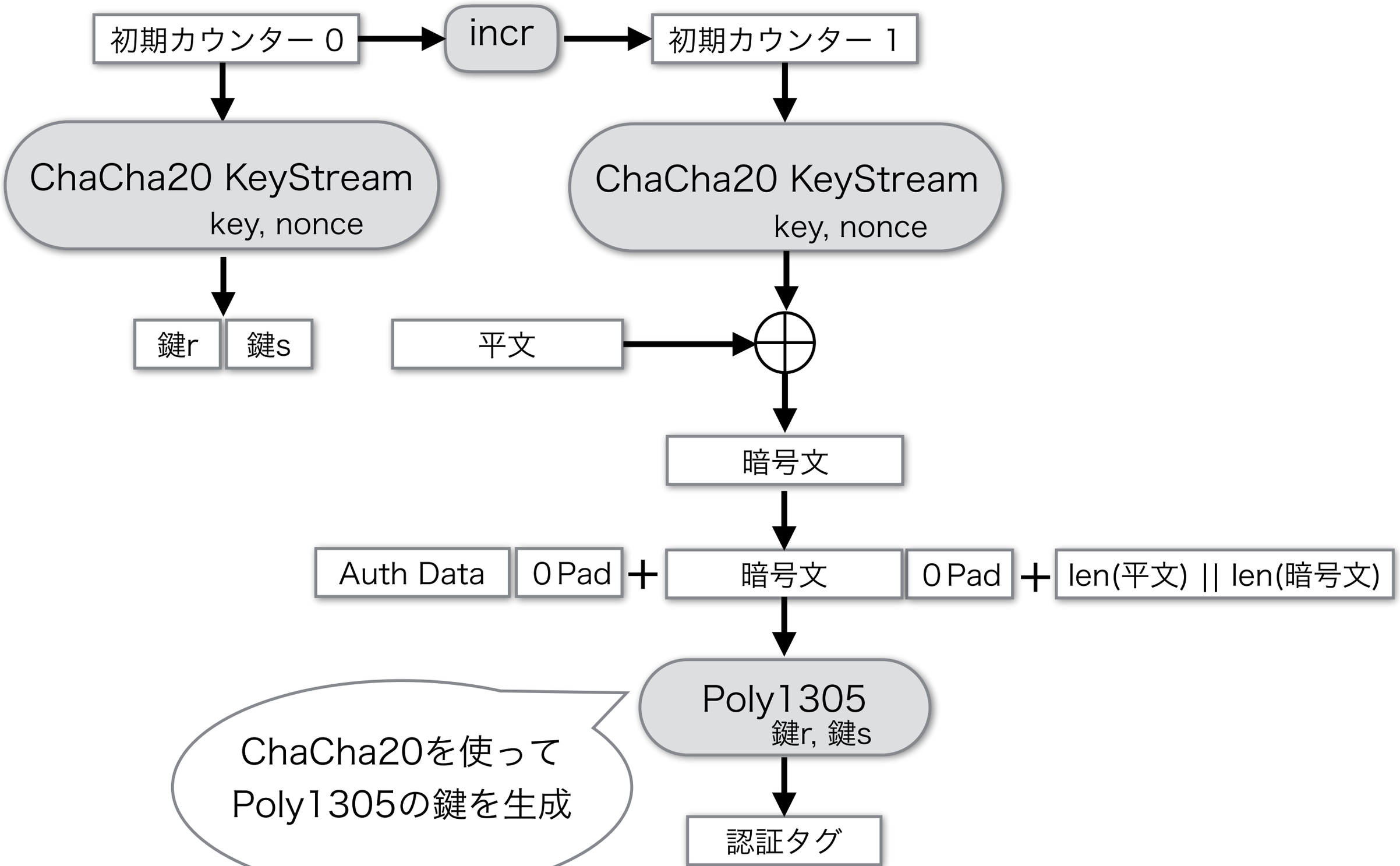
ChaCha20-Poly1305

を作る

TLS向けAES-GCMとChaCha20-Poly1305

	AES-GCM	ChaCha20-Poly1305
標準	RFC5288, RFC5289, RFC5487,	draft-ietf-tls-chacha20-poly1305-04 (2016/03/28時点IETF Last Call中)
対称暗号	AES128, AES256	ChaCha20
鍵交換	RSA, DH, DHE, ECDHE	ECDHE. DHE
認証	RSA, ECDSA, PSK	
PRF	SAH256(AES128), SHA386(AES256)	SHA256
明示的IV	8bytes	なし
Nonce	Client/Server WriteIV(4bytes) + 明示的IV(8bytes)	0パッドした Seq Num (12bytes) XOR Client/Server WriteIV(12bytes)
タグ長	16bytes	16bytes
最小暗号化長	25bytes	17bytes

ChaCha20-Poly1305によるAEAD生成



len(): 文字列長、64bit、little endian表記

演習

Poly1305 KeyGeneration

ChaCha20-Poly1305 Encryption

もし時間が余ったら

去年の演習やってみましょう。

sudo npm -g install seccamp2015-crypto-workshopper