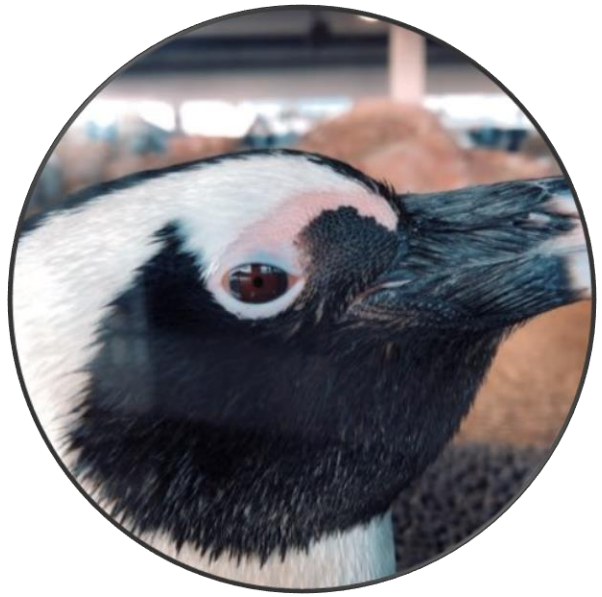
暗号目線で俯瞰するSSL/TLS

セキュリティ・ミニキャンプ 東京 2022

2022/12/18 うしがぃ

自己紹介



- うしがい@ushigai_sub
- 所属
  - 都立産業技術高等専門学校（5年）
  - SECCON Beginners 運営チーム
- 趣味
  - ピアノ：小6から8年間
  - ルービックキューブ：1/5/12/100=12.32/15.42/17.06/17.93
  - 将棋：アマ二段

内容

- SSL/TLSの基礎
  - イントロダクション
  - SSL/TLSで守りたいものは何か
  - 構成している標準化
  - 具体的な暗号アルゴリズム
- 暗号基礎
  - SSL/TLSで使用されている暗号技術の整理
  - [共通|公開]鍵暗号，鍵交換，乱数，ハッシュ，メッセージ認証，署名
- SSL/TLSとセキュリティ
  - SSL/TLSに存在する脅威
  - 脆弱性の例
  - 現行のバージョンに存在する脅威

到達目標

- SSL/TLSの通信がなぜ安全なのか

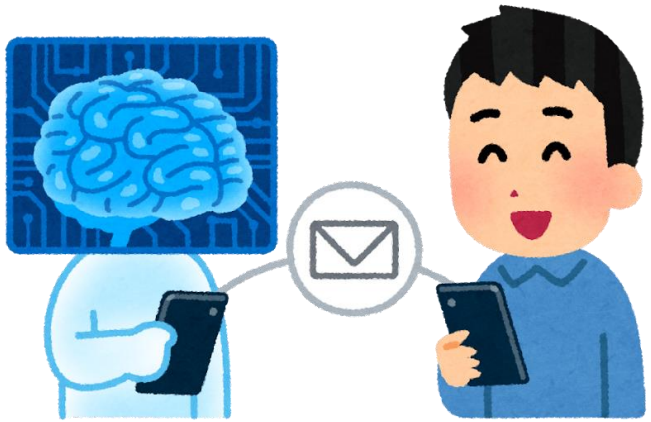
留意事項

- 講義中で使用するファイルはGitHubにおいてあります
- 適度に休憩を取ってください

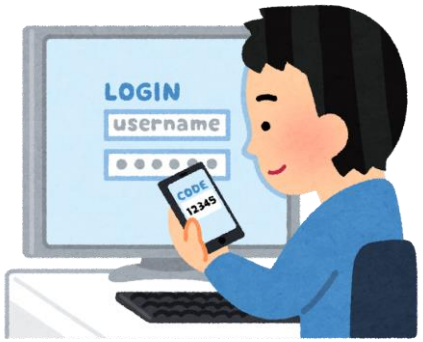
SSL/TLS

- HTTPSやSMTPSなどで使用されている暗号プロトコル

クレジットカード決済



メールの送受信



アカウントログイン

SSL/TLSという名称

- 1995年 SSL(Secure Sockets Layer)3.0が公開
  - Netscape Communications社が開発
- 1999年 TLS(Transport Layer Security)1.0の制定
  - 国際標準化のためにSSLから名称変更
  - 既にSSLという名称が定着していたためSSLという名前も残る
- SSLはTLSの前身で現在は使用されていない（プロトコルとしては）
- 本講義では`SSL/TLS`と表記し、バージョンを指定する場合は`SSL2.0`, `TLS1.2`のように表記

SSL/TLSで守りたいものはなにか

- JIS Q 27001での情報セキュリティの定義
  - 機密性，完全性，可用性
  - ただし真正性，責任追及性，信頼性，否認防止などの特性を維持することを含めることもある
- SSL/TLSの暗号技術では…
  - 機密性，完全性，真正性を保証する

機密性	データが通信途中で第三者により盗聴されても復号できないこと
完全性	データが通信途中で改ざんされていない状態を確保すること
真正性	通信相手がなりすましではなく確実に本人だと認証されていること

SSL/TLSに関係する標準化

SSL/TLSはセキュア通信プロトコルなので「プロトコル」と「暗号技術」に関するRFCが存在する

RFC番号	標準化プロトコル
6101	SSL3.0
2246	TLS1.0
4346	TLS1.1
5246	TLS1.2
8446	TLS1.3
9000	QUIC

SSL/TLSに関係する標準化

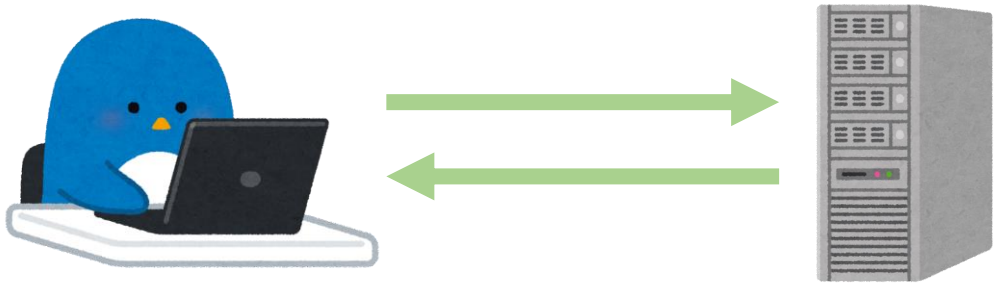
NISTによるSP-800, FIPS PUBシリーズのドキュメントも存在する

ドキュメント番号	関係する暗号技術
SP 800-90A	疑似乱数
SP 800-90B	疑似乱数のエントロピー（真性乱数）
FIPS PUB 186-4	DSA
SP 800-56A	DH/ECDH
FIPS PUB 197	AES
FIPS PUB 180-4	SHA-1, SHA-256, SHA-384, SHA-512
SP 800-38A	暗号利用モードCBC, CFB, CTR, OFB
SP 800-38C	暗号利用モードCCM
SP 800-38D	暗号利用モードGCM
SP 800-38B	CMAC
FIPS PUB 198-1	HMAC

SSL/TLSで使用されている暗号技術

1. ハンドシェイク
   - 1.1 通信先サーバの認証
   - 1.2 鍵共有アルゴリズム
2. 通信の暗号化
   - 2.1 共通鍵暗号による暗号化/復号
   - 2.2 HMAC（メッセージ認証符号）

ハンドシェイクとは……
サーバとクライアントの間で通信の暗号化時に使用する暗号スイートや各種パラメータの合意のこと



ハンドシェイク(TLS1.2)

1. Client Hello

2. Server Hello

3. Key Exchange

4. 暗号通信の開始

プリマスターシークレット

マスターシークレット

セッション鍵,IV,HMAC鍵

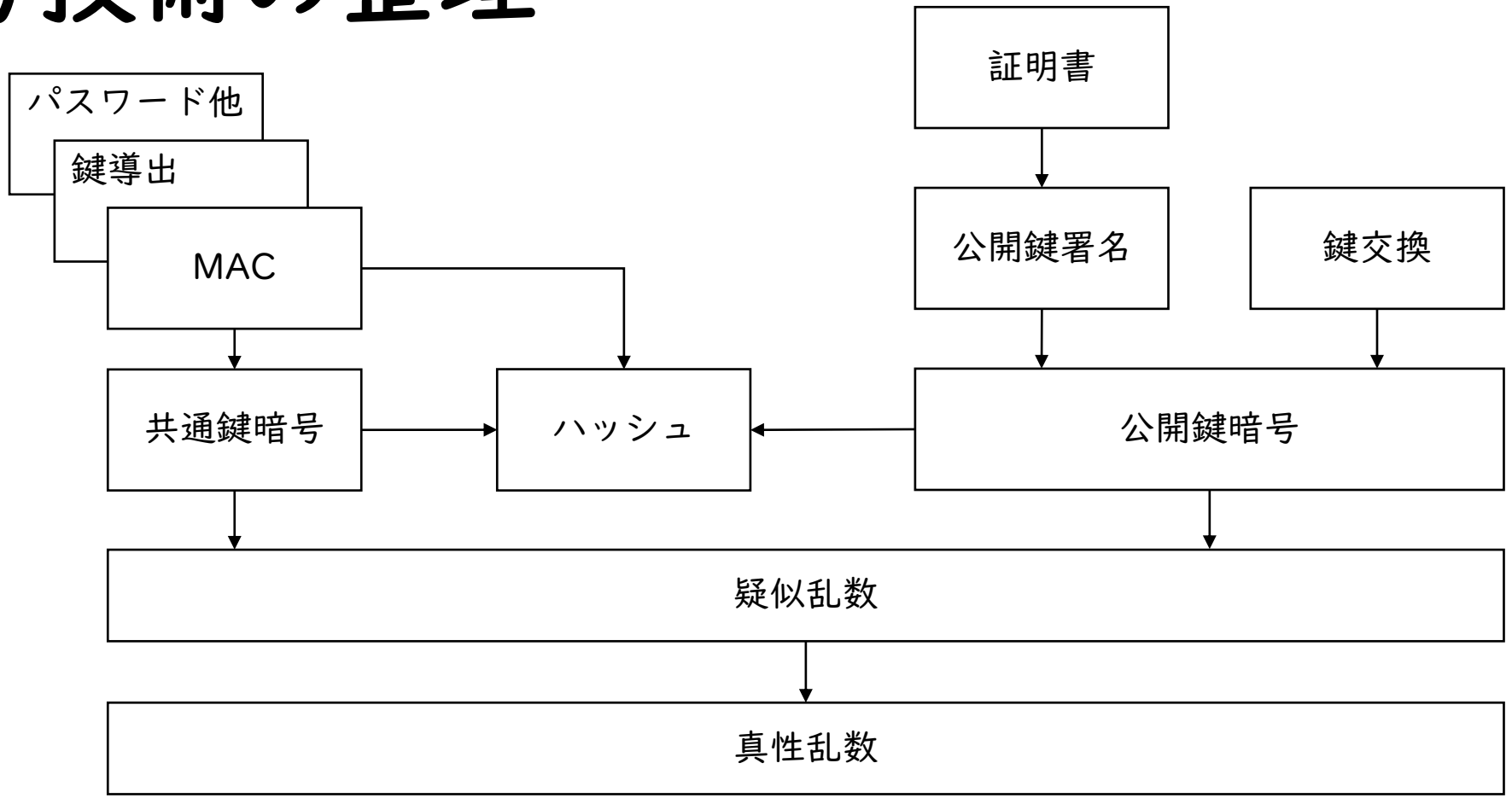
暗号スイート

opensslコマンドによる暗号スイートの確認方法を以下に示す
Transport Layer Security (TLS) Parameters (iana.org)も参照されたい

```
$ openssl ciphers -v
TLS_AES_256_GCM_SHA384 TLSv1.3 Kx=any Au=any Enc=AESGCM(256) Mac=AEAD
TLS_CHACHA20_POLY1305_SHA256 TLSv1.3 Kx=any Au=any Enc=CHACHA20/POLY1305(256)
Mac=AEAD
TLS_AES_128_GCM_SHA256 TLSv1.3 Kx=any Au=any Enc=AESGCM(128) Mac=AEAD
ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH Au=RSA Enc=AESGCM(256) Mac=AEAD
DHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=DH Au=RSA Enc=AESGCM(256) Mac=AEAD
ECDHE-ECDSA-CHACHA20-POLY1305 TLSv1.2 Kx=ECDH Au=ECDSA
Enc=CHACHA20/POLY1305(256) Mac=AEAD

  ~ snip ~
```

暗号技術の整理



参考『徹底解説TLS1.3（翔泳社）』p42 図3.1

暗号化方式

秘密鍵暗号化方式

暗号化と復号で**同じ鍵**を使い、

暗号化で使用する鍵を秘密にする方式

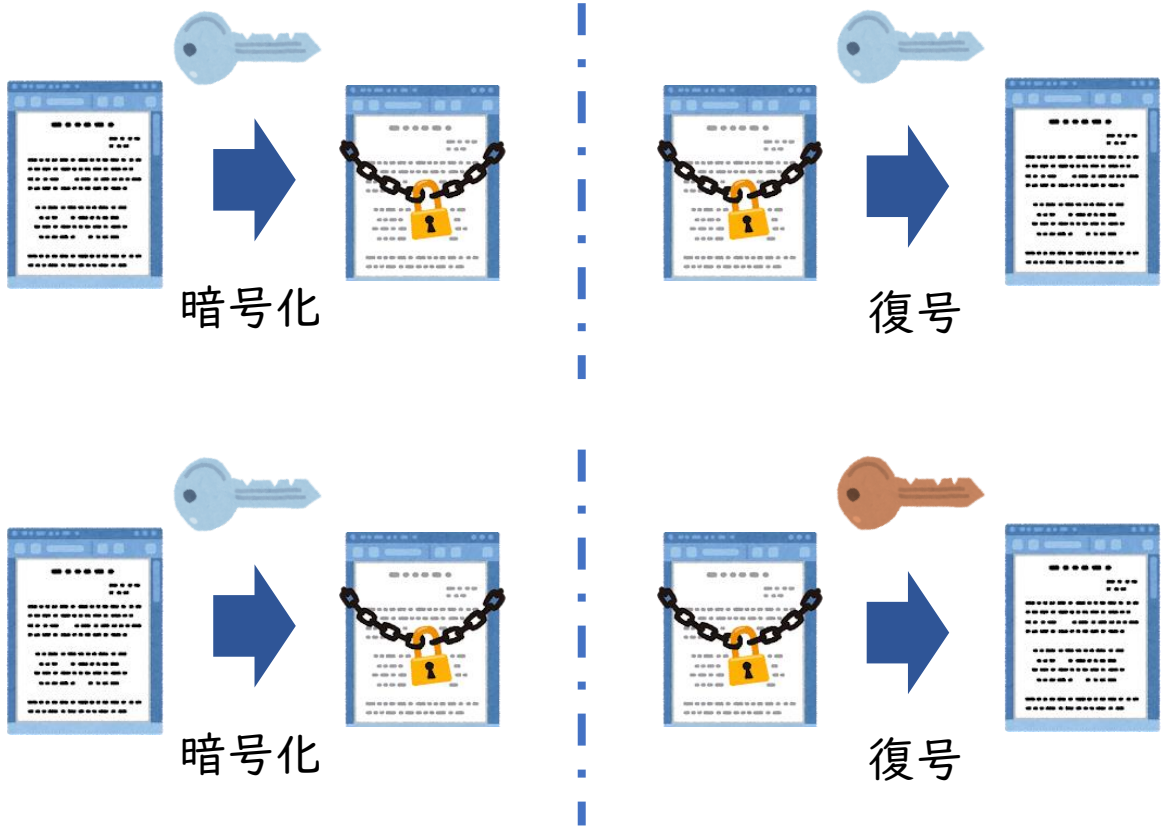
e.g. DES, AES, ChaCha20

公開鍵暗号化方式

暗号化と復号で**別の鍵**を使い、

暗号化で使用する鍵を公開する方式

e.g. RSA暗号, Rabin暗号



秘密鍵暗号化方式

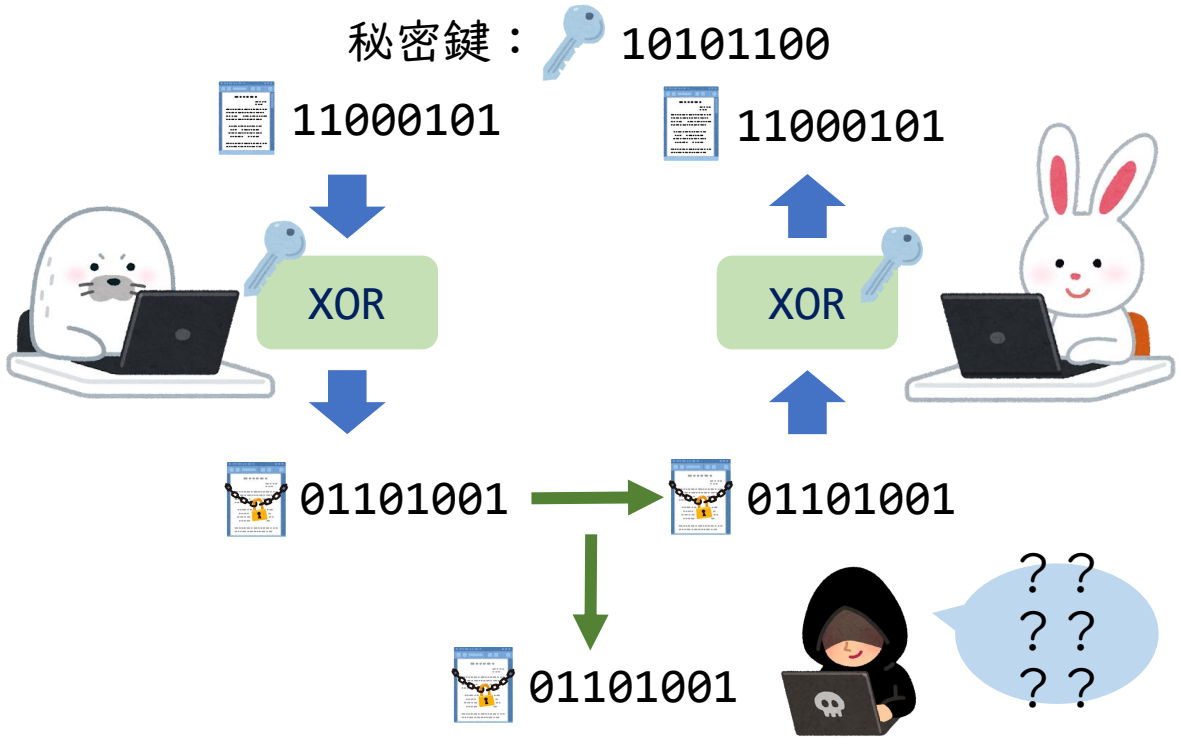
暗号化と復号で同じ鍵を使用する暗号化方式

共通鍵暗号化方式はストリーム暗号とブロック暗号に大別される

例：ワンタイムパッド

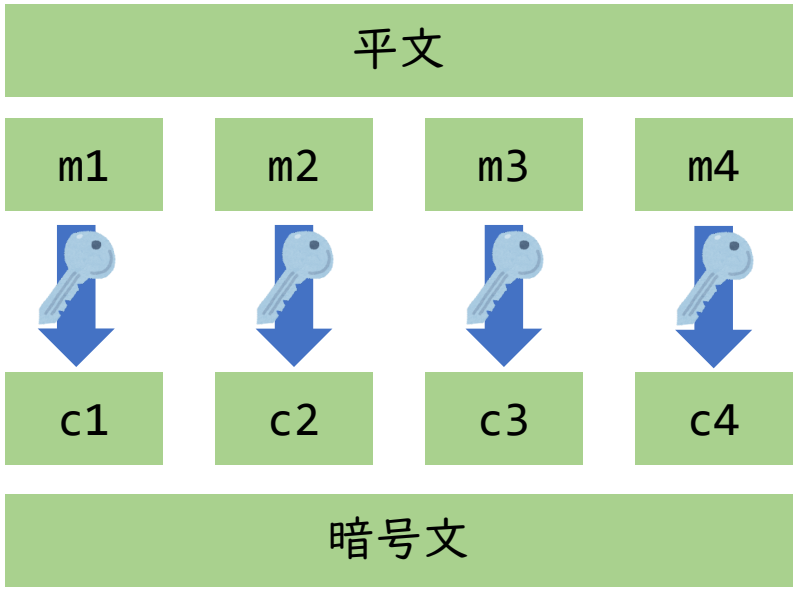
XORの真理値表

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

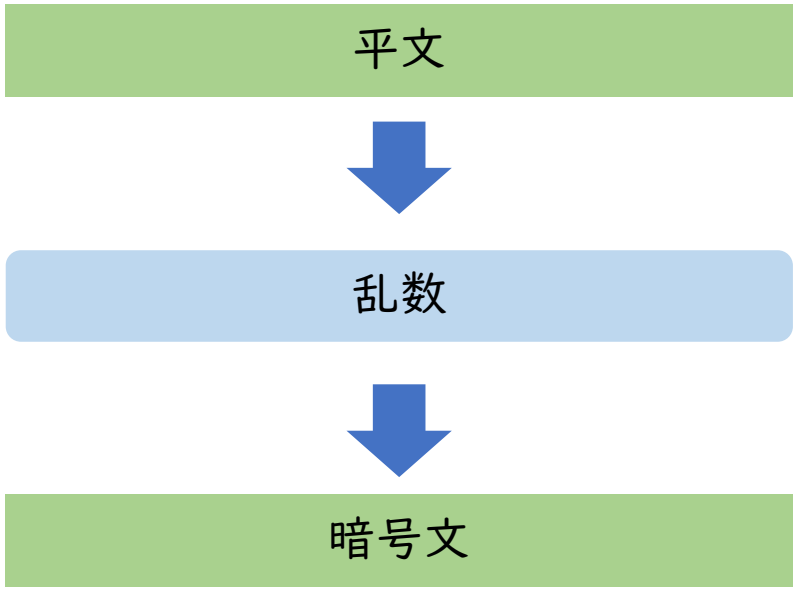


ストリーム暗号とブロック暗号

平文を固定長(ブロック)に分割し分割したブロックごとに暗号化を行う方式をブロック暗号といい、平文と同じ長さの乱数を生成し暗号化を行う方式をストリーム暗号という。



ブロック暗号



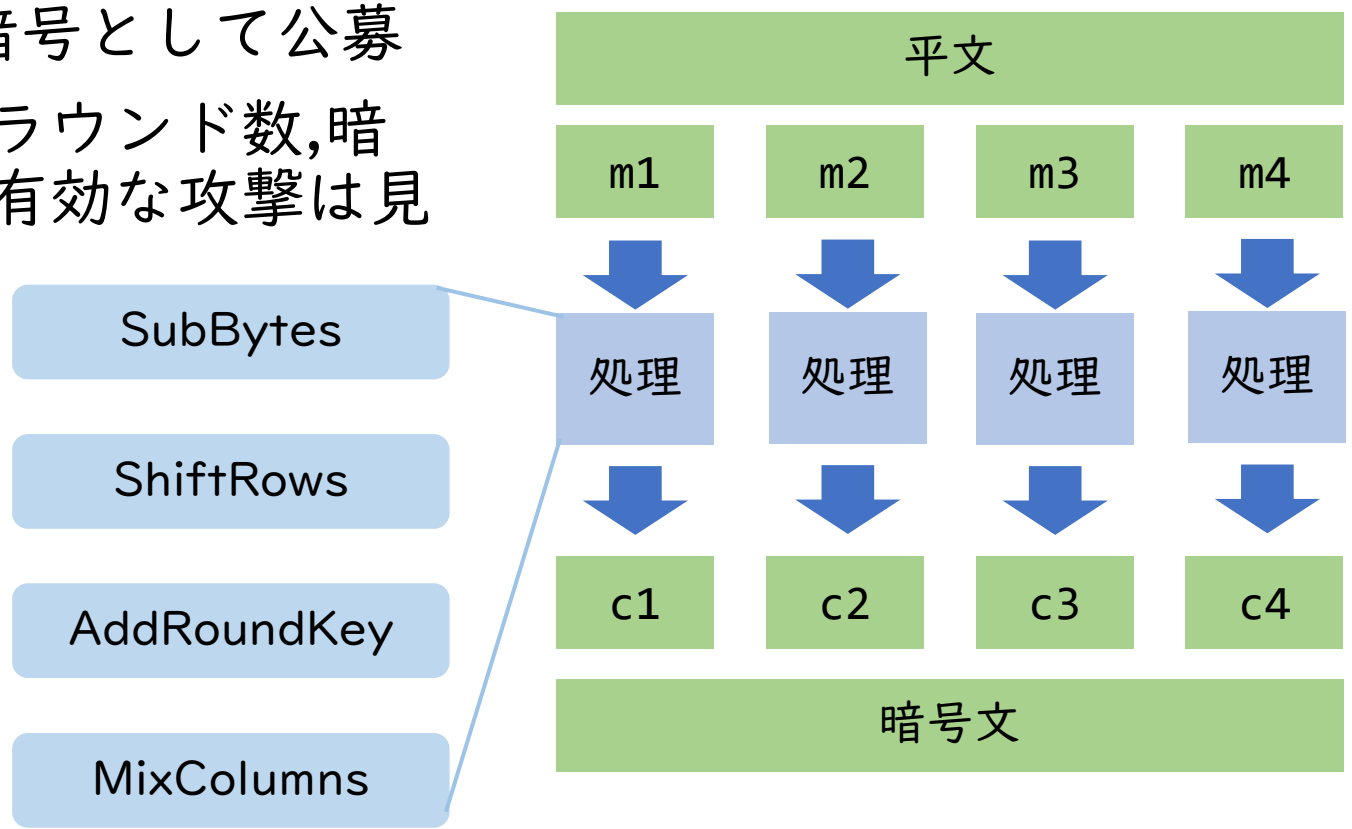
ストリーム暗号

AES(Advanced Encryption Standard)

- NISTがDESにとって代わる暗号として公募
- 2022年の時点で十分な鍵長,ラウンド数,暗号化モードを設定した場合の有効な攻撃は見つかっていない

平文をブロックに分割しそれぞれラウンド関数(SubBytes → ShiftRows → AddRoundKey → MixColumns)で暗号化を行う

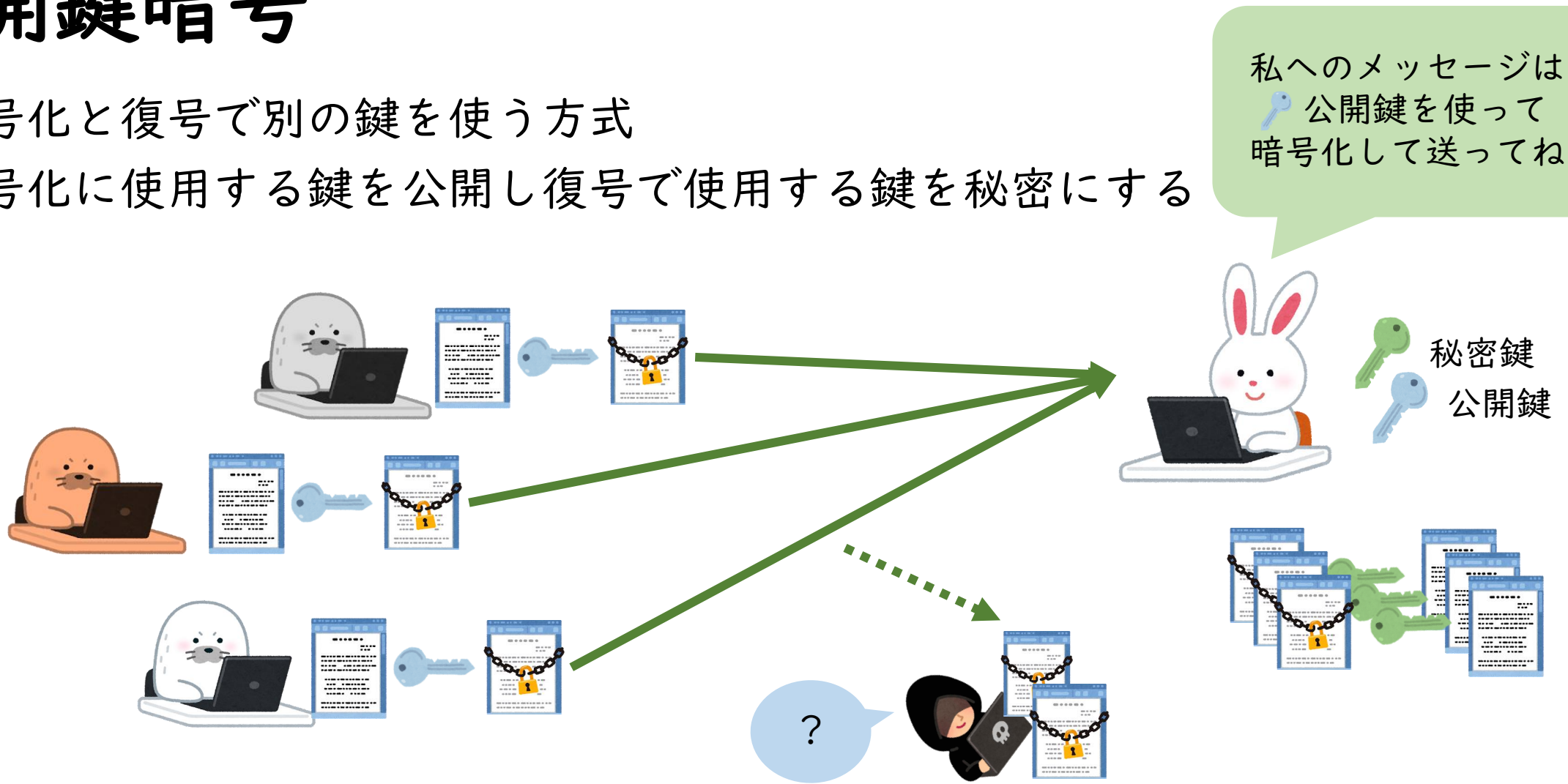
いくつかの暗号利用モードが定義されている(e.g. ECB,CBC,GCM, CCM, etc...)



公開鍵暗号

暗号化と復号で別の鍵を使う方式

暗号化に使用する鍵を公開し復号で使用する鍵を秘密にする



RSA暗号

- Rivest, Shamir, Adlemanによって発明
- 古参の公開鍵暗号
  - 1977年に発明
  - 公開鍵暗号を具体化した初の暗号化アルゴリズム
- SSL/TLSでは真正性を保証するためにデジタル署名で使用
  - TLS1.3では暗号化として使用されていないので注意
- 素因数分解が困難であることを安全性の根拠としている

公開鍵暗号/秘密鍵暗号（演習）

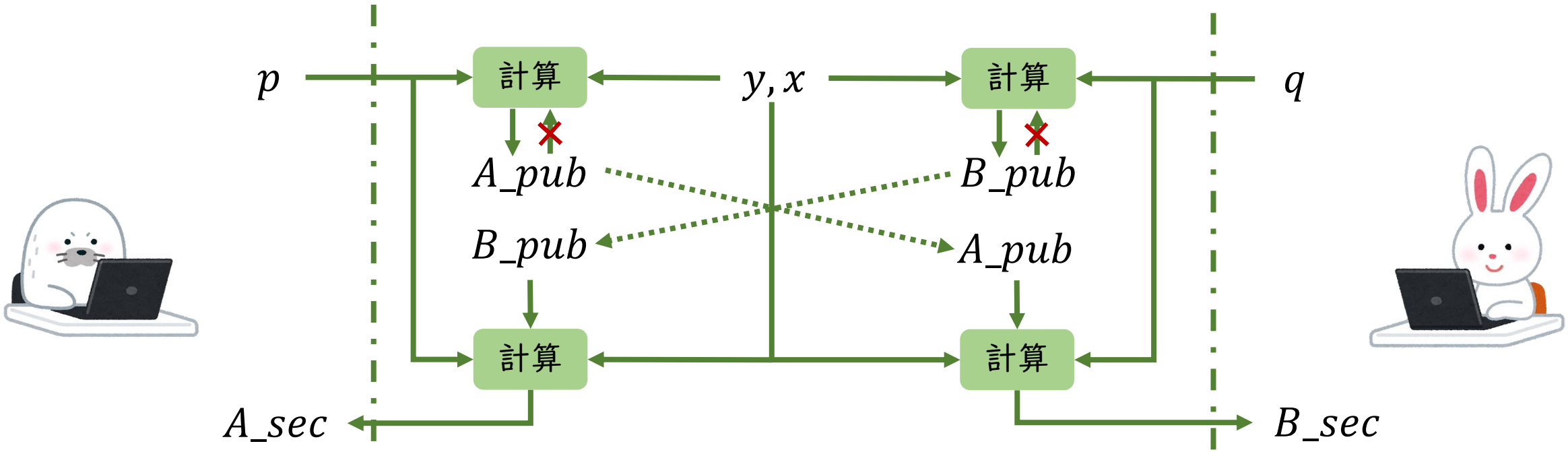
- `1_pubkey_seckey_encryption`ディレクトリにあるファイルを実行してみよう
- Q. RSA暗号において公開&秘密にされている変数はなにか
- Q. どちらの暗号化方式のほうが高速か
- Q. 大量のデータを送信する場合どちらが適しているか

鍵交換アルゴリズム

暗号化した通信の送受信において鍵を盗聴されることなく共有したい

DH/ECDH

数学を利用した鍵交換アルゴリズムとしてDH/ECDH鍵共有がある。以下にDH鍵共有の鍵共有までの模式図を示す。

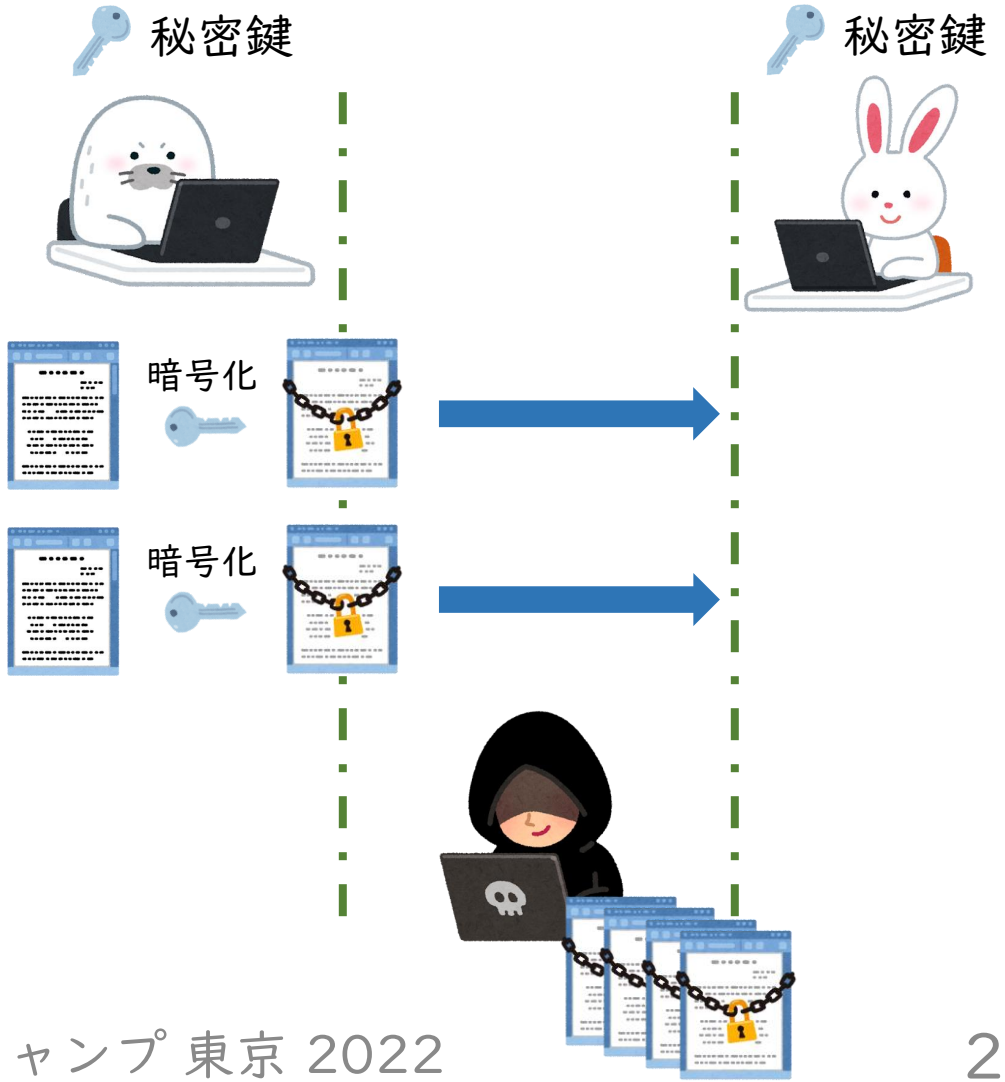


鍵交換アルゴリズム（演習）

- 2_key_exchangeにあるファイルを実行してみよう
- 1ページ前のスライドでの 計算 の処理は実際どのような計算をしているだろうか

鍵交換とPFS

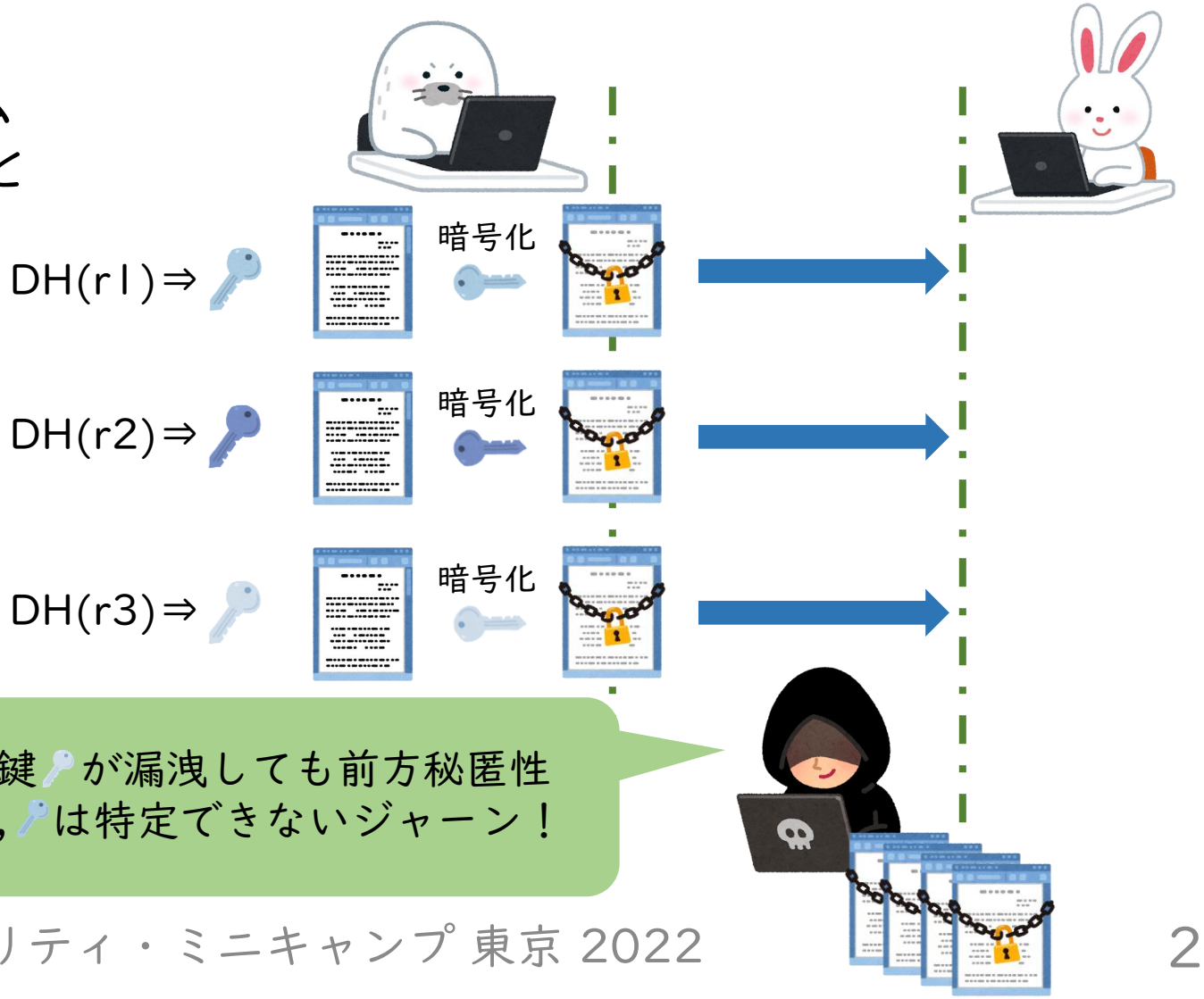
- 右図では常に同じ鍵で暗号化したメッセージのやり取りをしている
- 今までの暗号文を盗聴している攻撃者が秘密鍵を手に入れた場合、暗号文をすべて復号されるリスクが存在する
- PFS(Perfect Forward Secrecy)とは鍵交換で共有した秘密鍵と暗号文が盗聴されても、盗聴者は暗号文を復号できないという概念のことである



鍵交換とPFS

PFSをもつ鍵共有アルゴリズムをDHE(Ephemeral)/ECDHEとよぶ

TLS1.2では鍵交換アルゴリズムとしてDH/ECDHや静的RSAが採用されていた。しかしTLS1.3では完全前方秘匿性を確保するためにそれらは廃止されDHE/ECDHEのみが採用された。



乱数

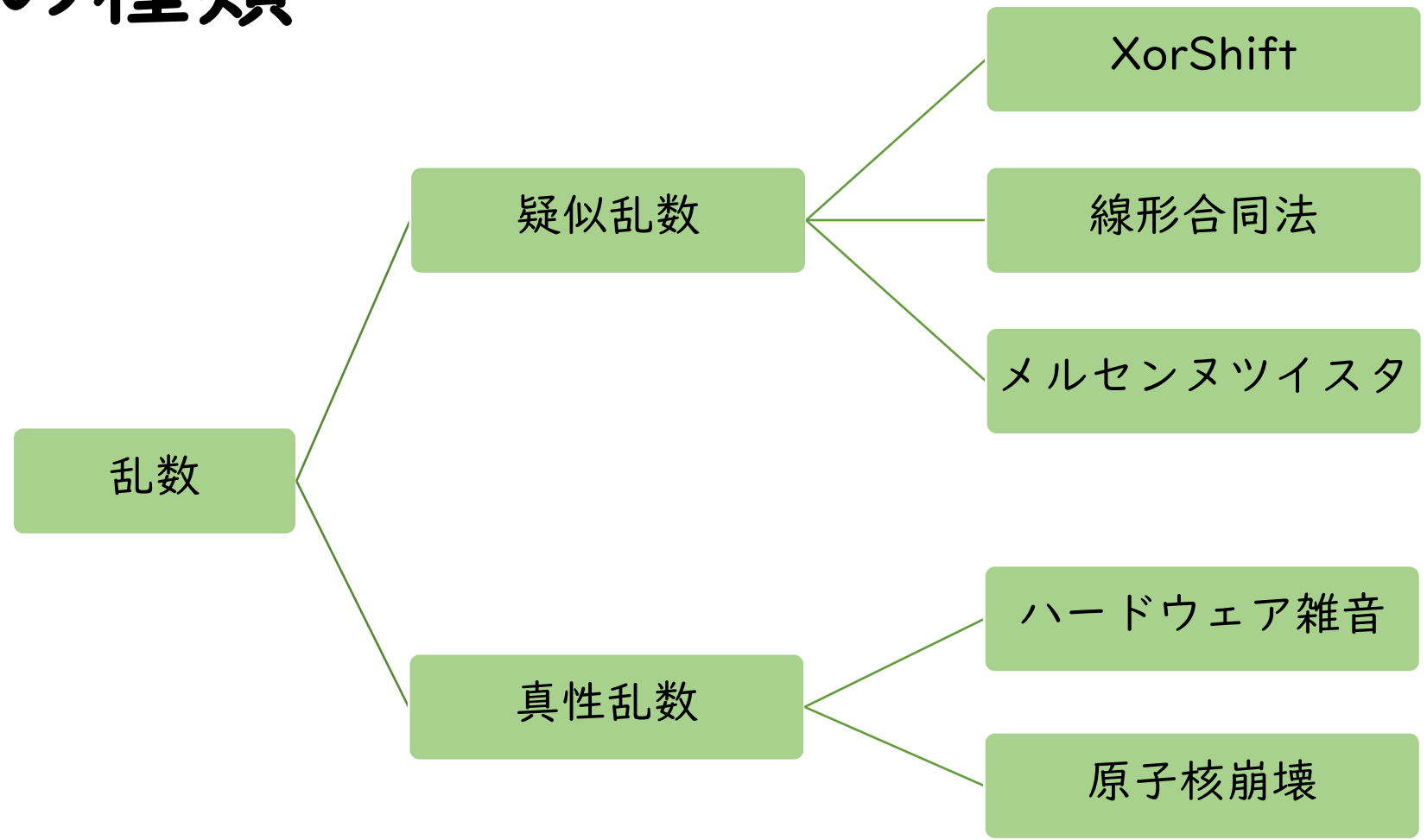
現実世界で乱数が使用される場面は？

- 公開鍵/秘密鍵
- Webサイトのセッションキー
- ソシャゲのガチャ, エンカウント乱数
- 統計処理の研究

それら乱数に対する要件は？

- 高速かつ軽量に生成できること
- 統計的に無作為であること
- 過去の出力から未来の出力を予測できないこと

乱数の種類



真性乱数生成器(TRNG)

ハードウェアの熱雑音や原子核崩壊などを利用した予測できない乱数。私たちが使用しているPCのプロセッサには乱数生成するための機能が提供されており、各種プログラミング言語から簡単にアクセスできる。

- メリット
  - 非決定的な方法で生成するので次の値を予測できず真にランダムな値を提供できる
- デメリット
  - 生成に時間がかかる
  - サイドチャネル攻撃などによりバイアスがかかる可能性がある

疑似乱数生成器（PRNG）

真性乱数は予測できないがハードウェアの状態をわざわざ観測しているので生成に時間がかかる。特に原子力シミュレーションには数十億の乱数を消費する。乱数を高速に生成するためには疑似乱数生成器を使用する。

疑似乱数生成器の例

- XorShift
- 線形合同法
- LFSR
- メルセンヌツイスタ

疑似乱数生成器は決定的アルゴリズムで生成されるので内部状態を観測出来たら予測できるよ。
乱数生成器の品質測定のための統計的テストとしてダイハードテストなどがあるよ。

PRNGの例

線形合同法（LCG）

ある自然数A, B, Mに対し漸化式$X_{n+1} = A \times X_n + B \pmod{M}$で与えられる疑似乱数生成器である。

例えば$A = 7, B = 6, M = 11, X_0 = 4$とすると

$X_1 = A \times X_0 + B \pmod{M} = 7 \times 4 + 6 \pmod{11} = 1$

$X_2 = A \times X_1 + B \pmod{M} = 7 \times 1 + 6 \pmod{11} = 2$

$X_3 = A \times X_2 + B \pmod{M} = 7 \times 2 + 6 \pmod{11} = 9$

のようにその都度計算していけば乱数を生成できる。

ただし線形合同法は周期が高々Mでまた6つ連続した乱数を得られればそれ以降の乱数を予測することができる。

→ メルセンヌツイスタを使用するのが一般的

暗号論的疑似乱数(CSPRNG)

疑似乱数生成器はハードウェアによる真性乱数生成器と比べ高速に動作するが前述のとおり過去の出力をもとに予測できてしまう可能性がある。

- PRNGは出力から内部状態を復元できてしまうのでCSPRNGには出力する前にハッシュ関数や共通鍵暗号を使用している
- Linuxの場合`/dev/random`や`/dev/urandom`はTRNGをエントロピーとし、ストリーム暗号によって乱数を生成するCSPRNGである

決定的に生成しつつも予測される危険性を不可逆な処理によって予測困難にしているね
TLSでの鍵生成などもこのCSPRNGが使用されているよ

乱数（演習）

- pythonの`random`モジュールや`secrets`モジュールを動かしてみよう
- Q. 疑似乱数(randomモジュール)と暗号論的疑似乱数生成器(secretsモジュール)はどちらが高速か
- Q. （Linuxを使用している人向け）`/dev/random`を使用して乱数を生成してみよう

乱数の内部状態ついて

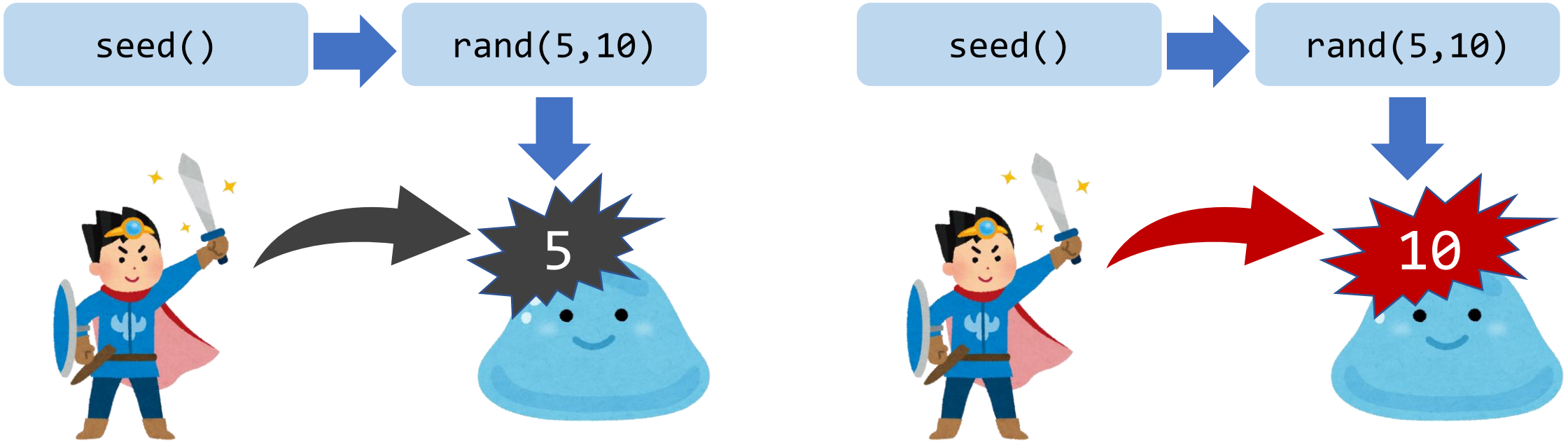
疑似乱数生成期なら乱数の内部状態の
SEED値を保存すれば乱数の再現が可能

Q.乱数のSEED値の初期値は何が適切か？

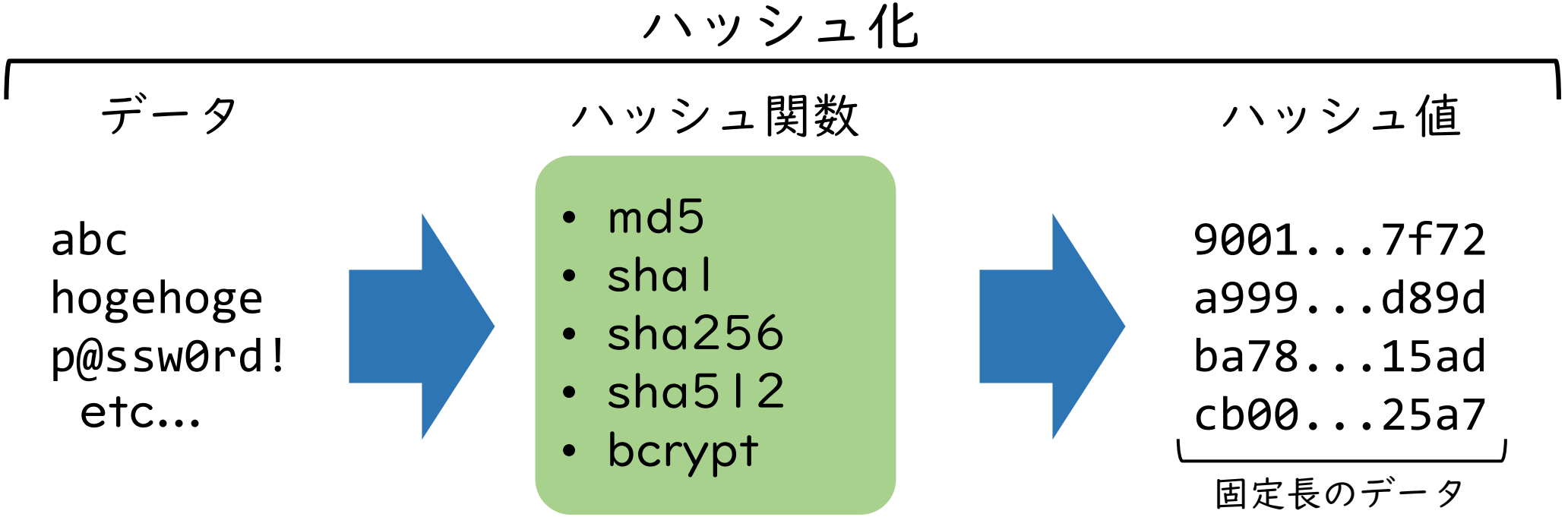
```
>>> import random
>>> random.seed(20221218)
>>> random.random()
0.273832247748559 5
>>> random.random()
0.452036229831616 4
>>> random.random()
0.582924144054909 8
>>> random.seed(20221218)
>>> random.random()
0.273832247748559 5
>>> random.random()
0.452036229831616 4
>>> random.random()
0.582924144054909 8
>>>
```

余談ゲームでの乱数調整について

ゲーム(TAS)における乱数調整とは



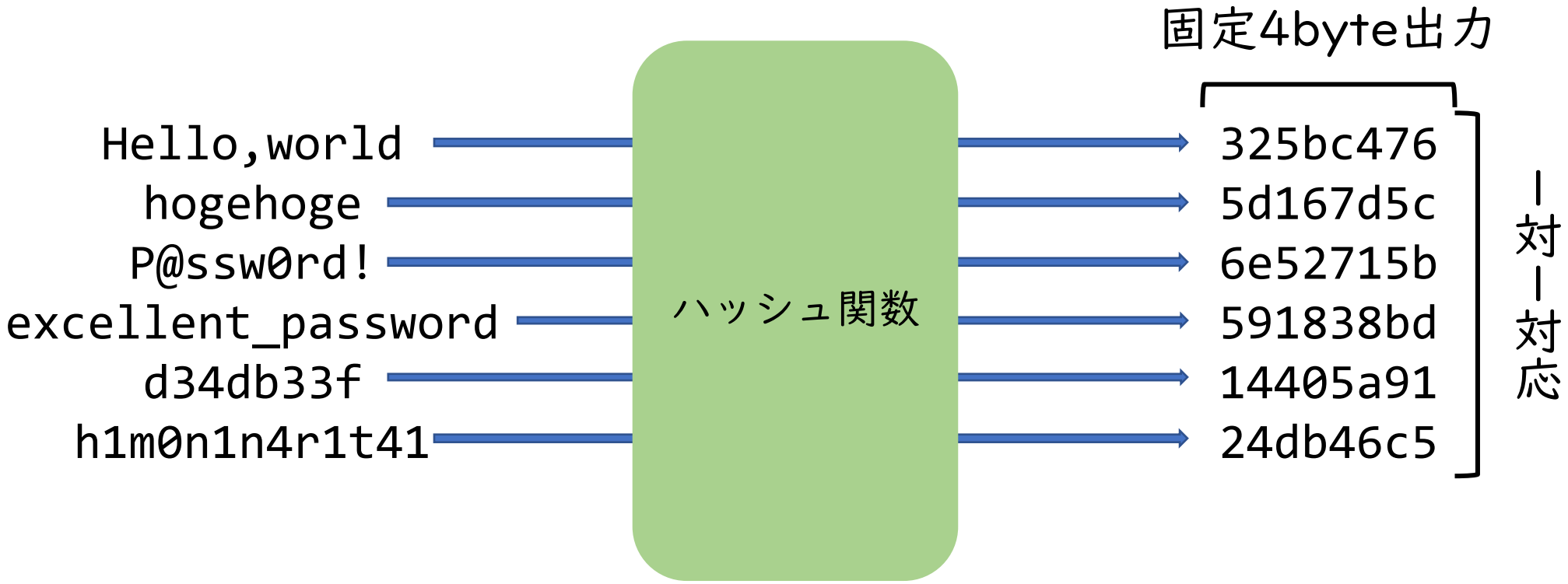
ハッシュ



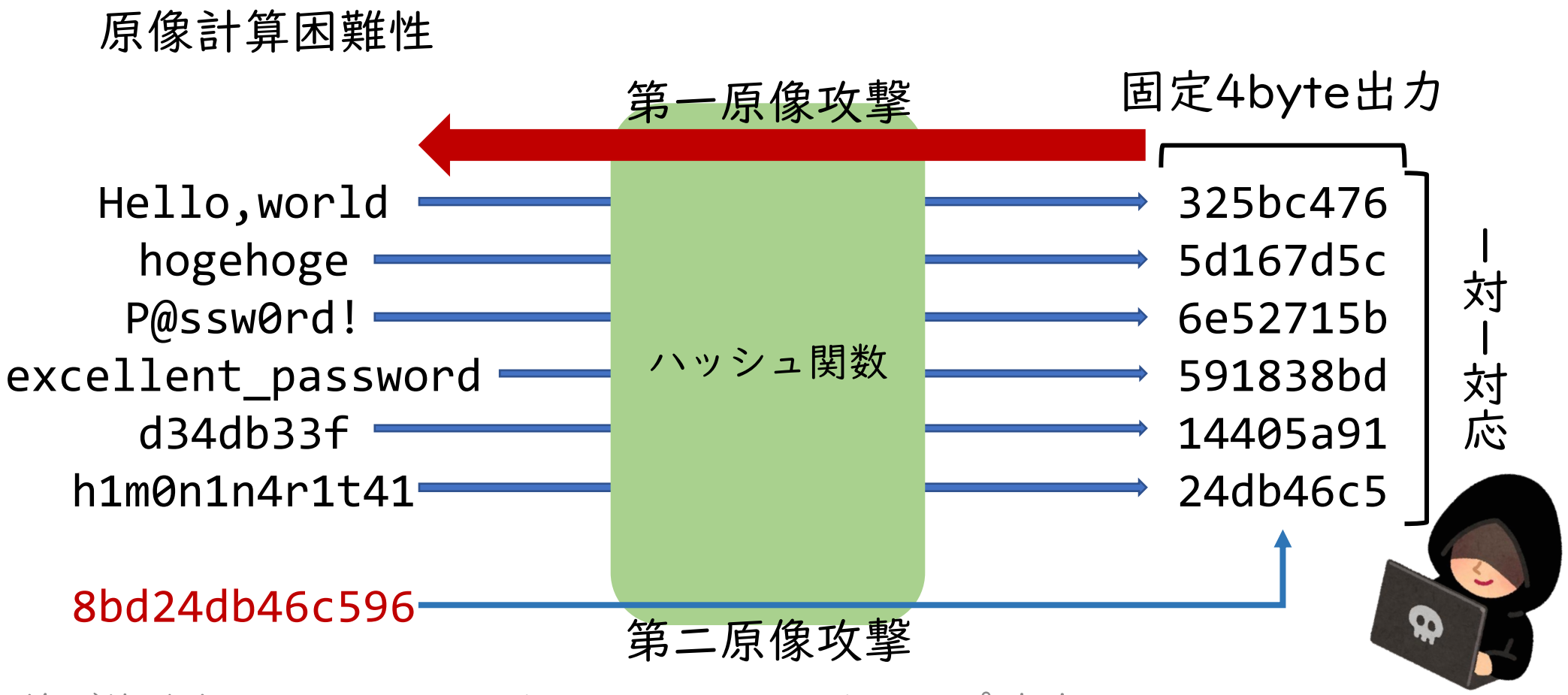
データ`Hello world!!!`をハッシュ関数sha1でハッシュ化する場合、
ハッシュ値は`6555aa9d245f6dc2b57aa13366cc6c6fcccab6ad`となる

ハッシュの特性 (1/2)

決定的で固定長の出力

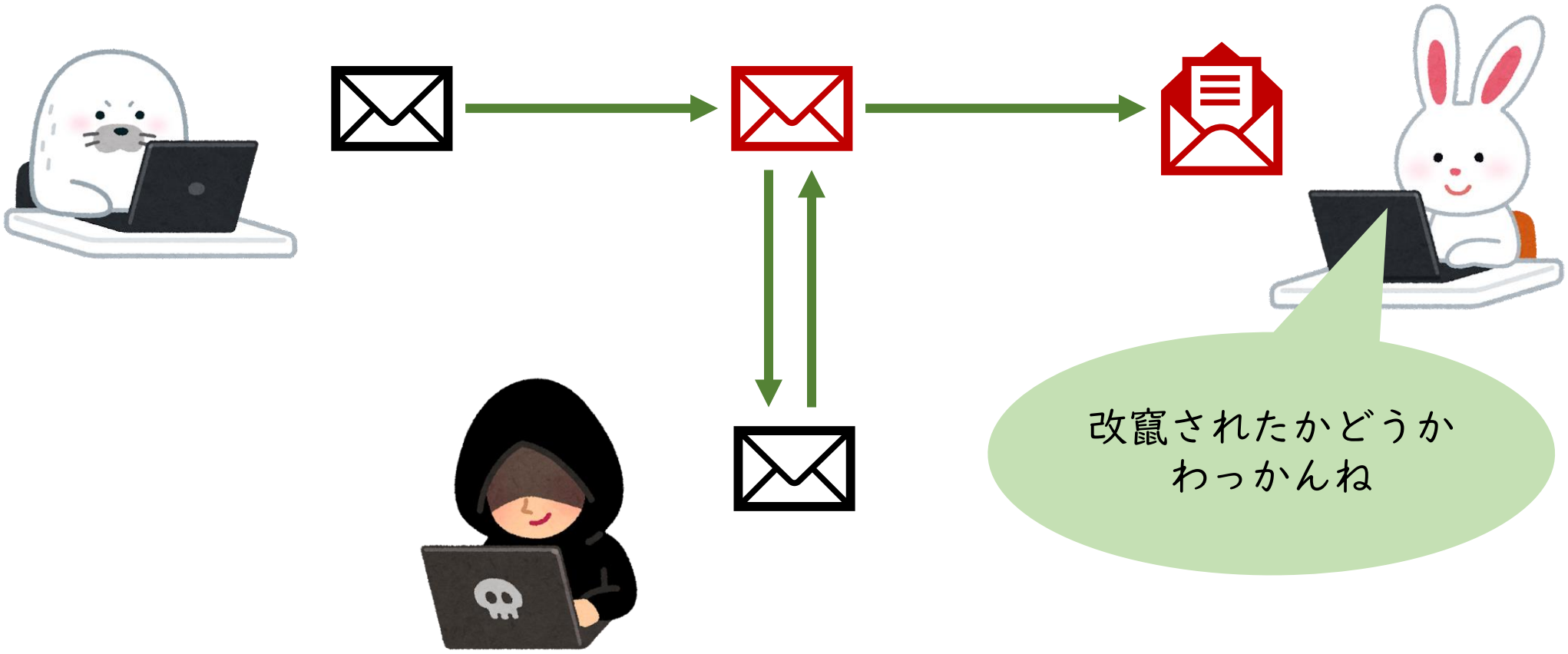


ハッシュの特性 (2/2)

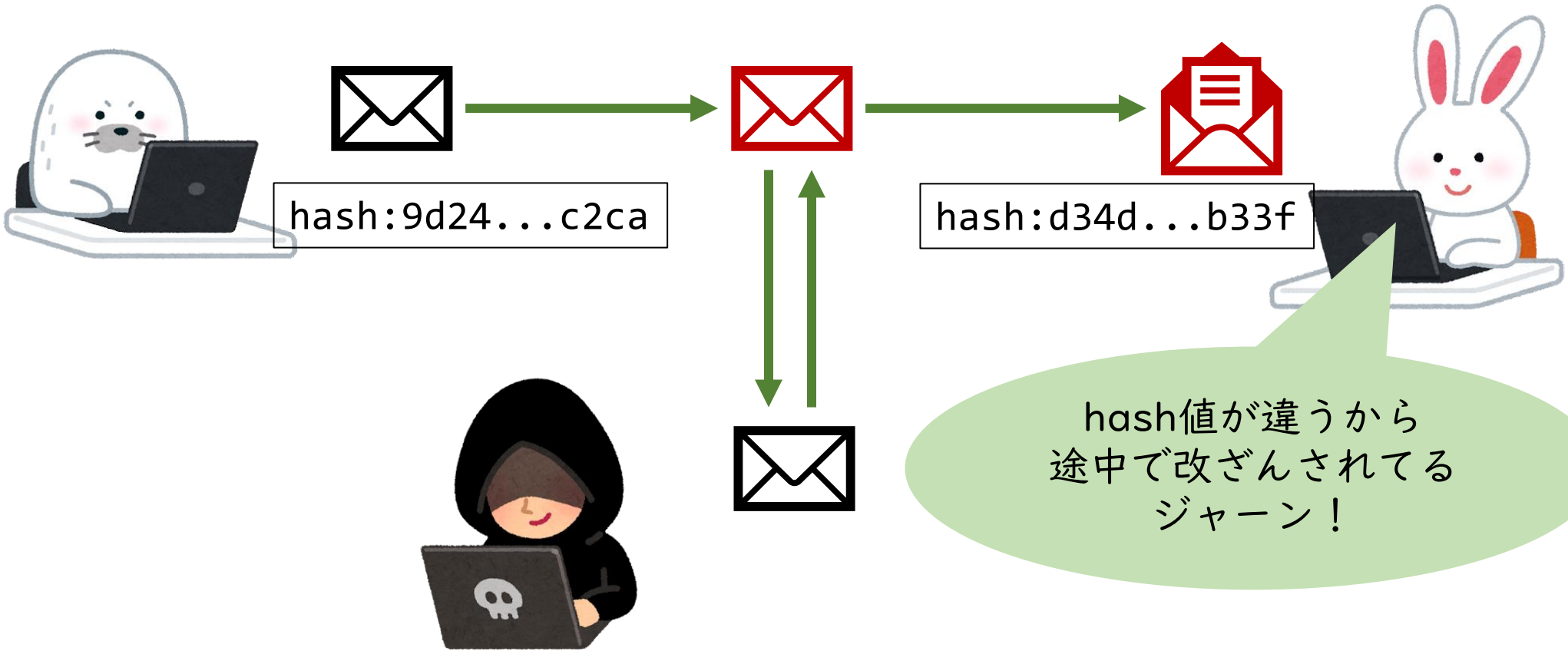

原像計算困難性
第一原像攻撃
固定4byte出力
Hello,world
hogehoge
P@ssw0rd!
excellent_password
d34db33f
h1m0n1n4r1t41
ハッシュ関数
325bc476
5d167d5c
6e52715b
591838bd
14405a91
24db46c5
一対一対応
8bd24db46c596
第二原像攻撃

ハッシュがどこで使われているのか

ハッシュがどこで使われているのか



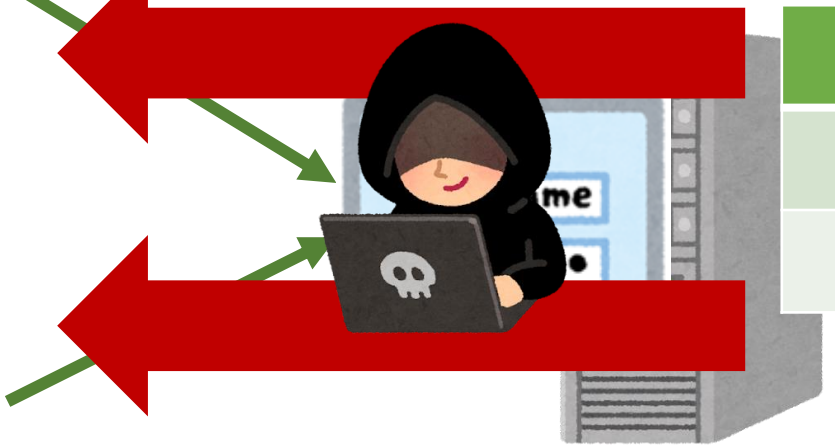
ハッシュがどこで使われているのか



ハッシュがどこで使われているのか

name:usagi
pass:usausa123

name:azarashi
pass:p@ssw0rd!



username	password
usagi	usausa123
azarashi	p@ssw0rd!

ログイン情報をもつテーブルが流出したときに
攻撃者はパスワードを入手できてしまう

ハッシュがどこで使われているのか

name:usagi
pass:usausa123

name:azarashi
pass:p@ssw0rd!

sha256()

username	password
usagi	568a...4de8
azarashi	2c01...5814

hashは一方向性だから
hash値だけ入手しても分
かんないジャーン！

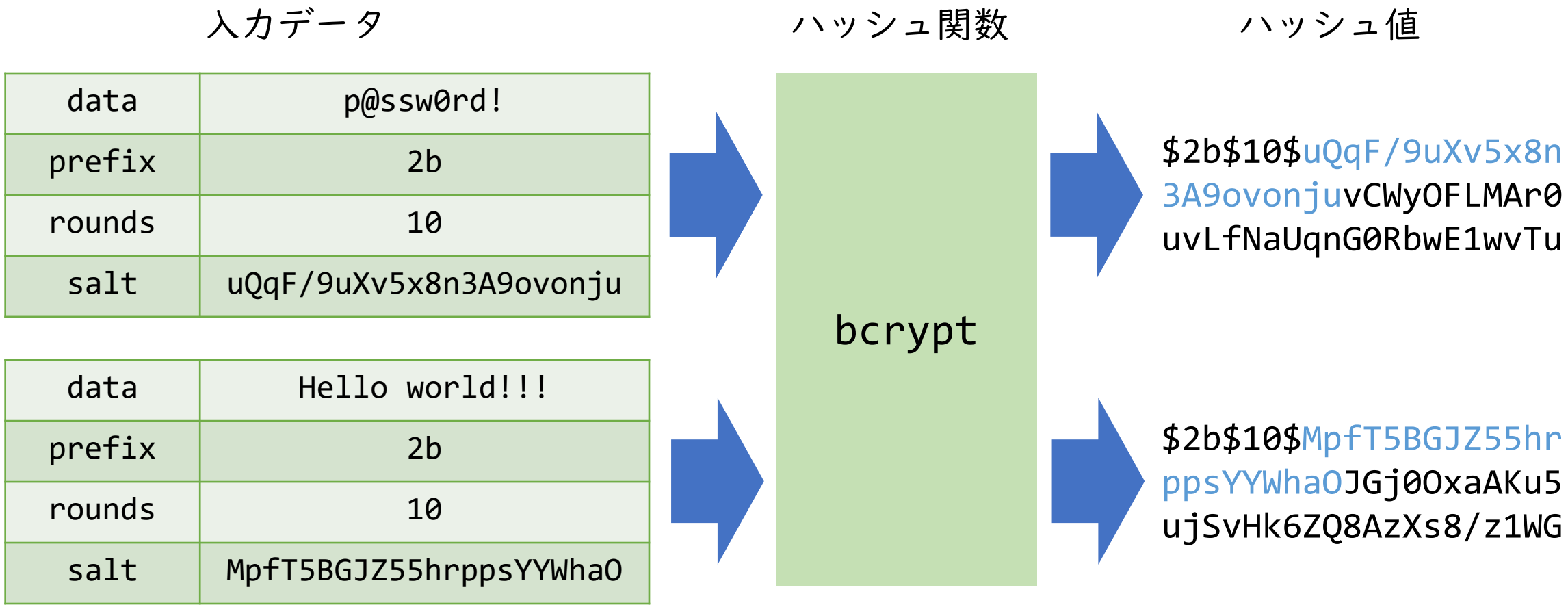
568a...4de8

2c01...5814

ソルトとストレッチング

- ハッシュ値をより強固にするためのもの
- ハッシュ化したいdata以外にprefix, rounds, saltが必要
- prefix
  - ハッシュ化を行うbcryptのバージョンで2a, 2x, 2y, 2bのように指定できる
- salt
  - 22文字（128bit）のランダムな文字列
  - dataが同じでもsaltが異なることで全く違うハッシュ値が出力される
- rounds
  - ストレッチング（ハッシュ値のハッシュ値を取る）回数を表し例えば5と指定すれば2^5=32回ストレッチングを行う
  - NISTよれば2^10=1024回ストレッチングを行うことが推奨されている

ソルトとストレッチング



演習（ハッシュ）

- `4_hash_function`ディレクトリにあるファイルを実行してみてください
- `bcrypt`がソルトによって違うハッシュ値を生成することを確かめてみてください

鍵導出

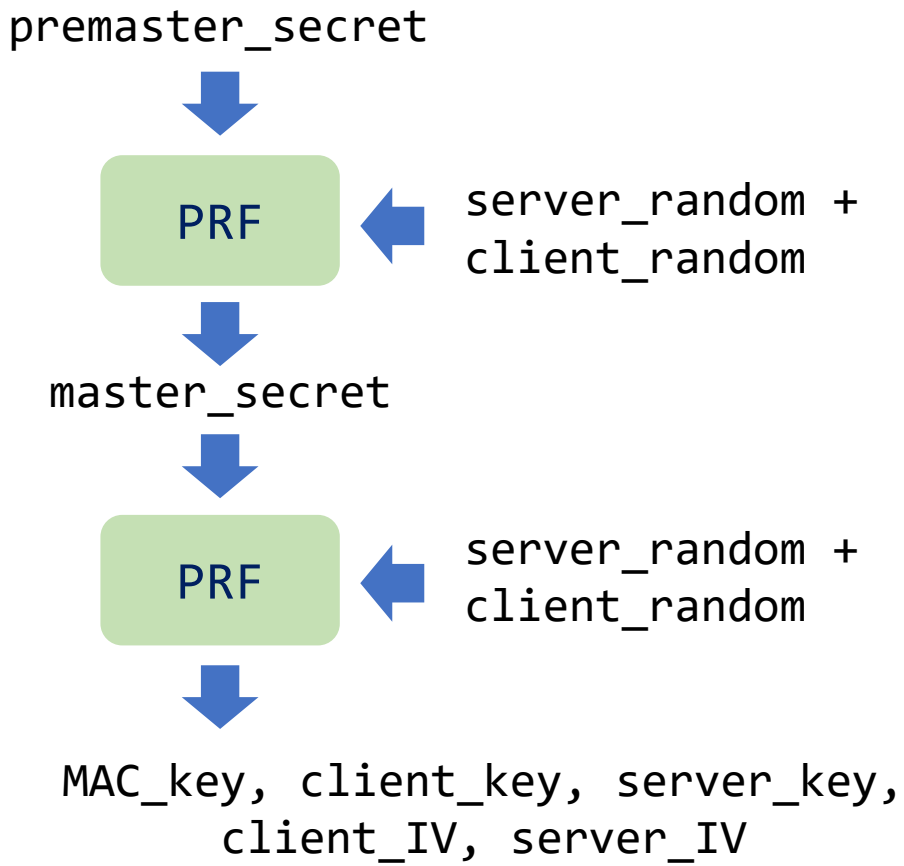
- 安全性を高めるために鍵共有で共有した鍵をそのまま使用せず、鍵導出関数に渡しその出力を秘密鍵として使用
- 鍵交換アルゴリズムを使用してプリマスターシークレットという同じ値を共有
- プリマスターシークレットと鍵導出関数を使用してマスターシークレットやセッション鍵（共通鍵）やIVなどを生成

TLS1.2の鍵導出関数

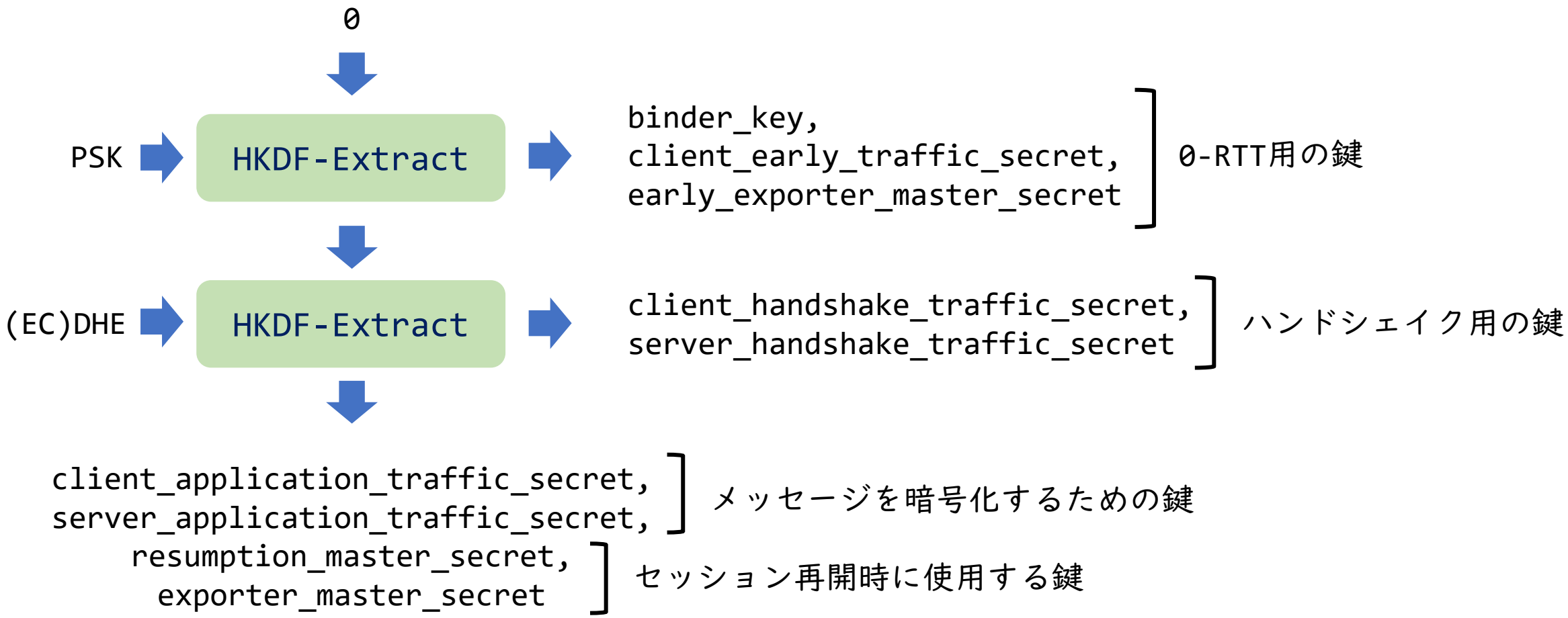
TLS1.2ではPRF(Pseudo Random Function)が使用されており、`master_secret`及び暗号化で使用する各種パラメータの生成に使用されている

PRFは48bytesの出力をもち、鍵生成での呼び出し時には、要求される長さになるまでPRFを呼び出し結果を連結させる

ハンドシェイクでの中間者攻撃(RacconAttack, Logjam Attackなど)でセッション鍵が予測される攻撃が発表され、TLS1.3では別の鍵導出関数が使用されている



TLS1.3の鍵導出関数

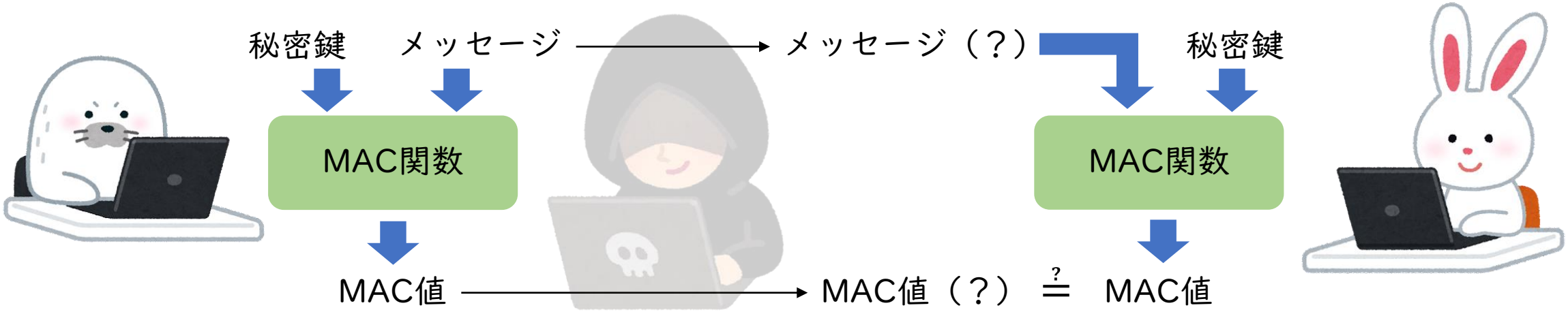


鍵導出（演習）

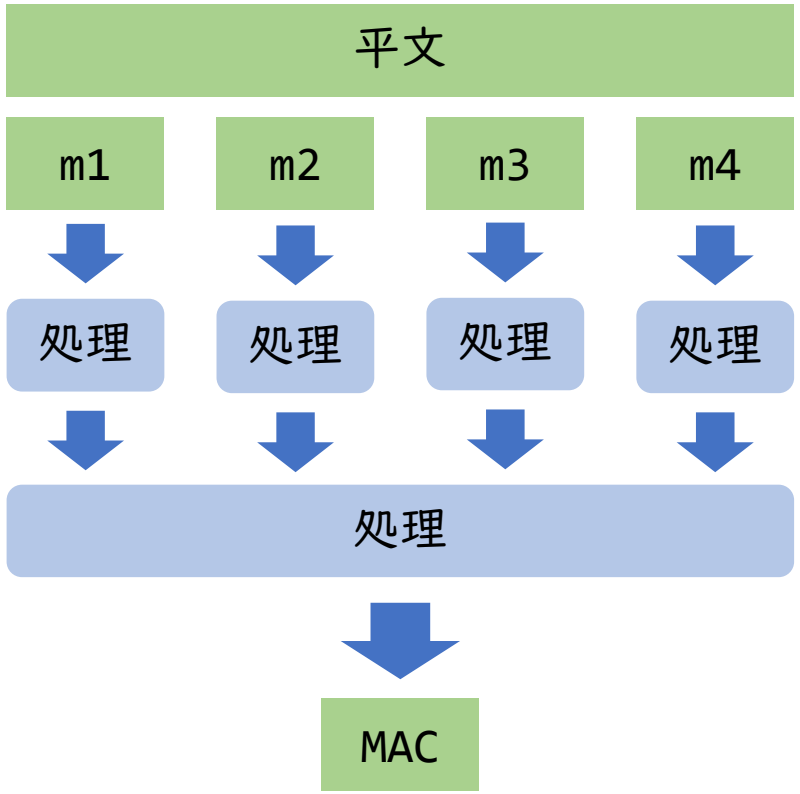
- `5_hkdf`ディレクトリにあるファイルを実行してみよう
- `test_case`を通過させてみよう

メッセージ認証

メッセージ認証符号（MAC:Message Authentication Code）とは通信が改ざんされていないかどうかを検知し、完全性を保証する暗号アルゴリズム

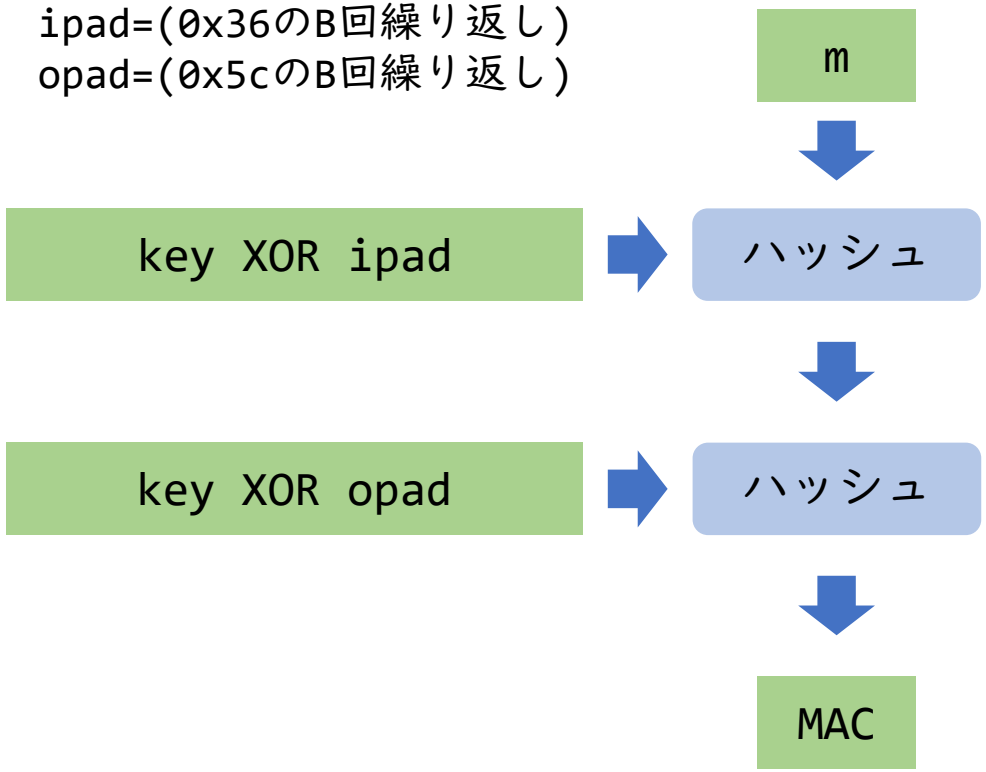


MACの方式



ブロック暗号を使用したMAC関数の概略図

B=hash関数の出力bytes長
ipad=(0x36のB回繰り返し)
opad=(0x5cのB回繰り返し)



HMACの概略図

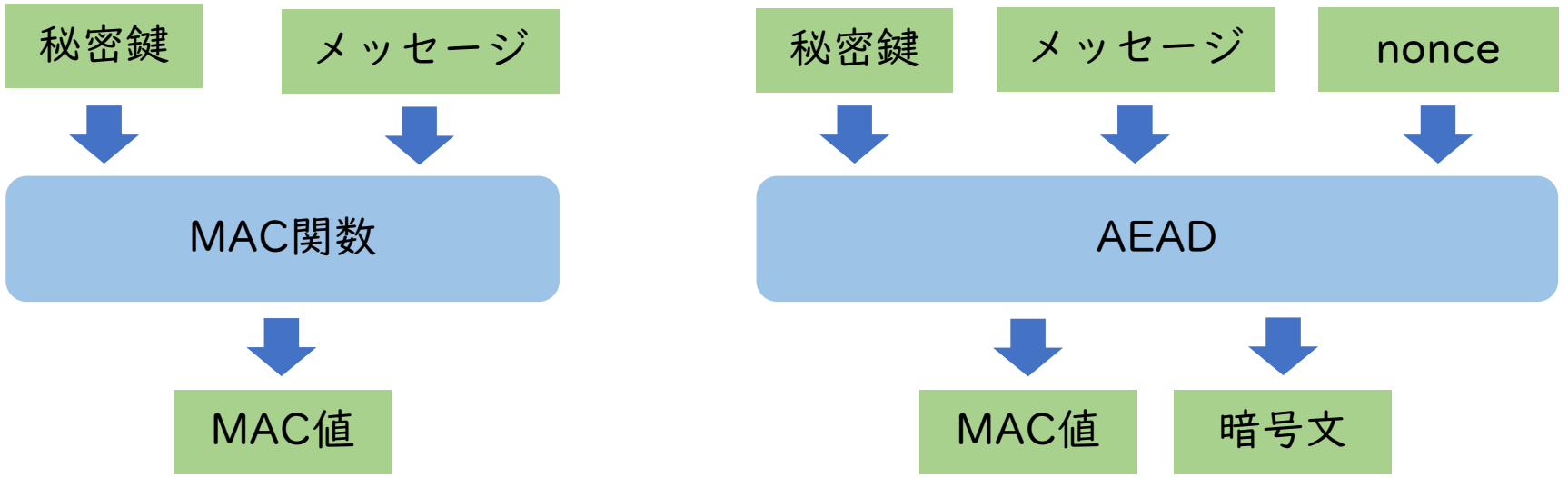
MAC（演習）

- `6_message_authentication`にあるファイルを実行してみよう
- `test_case`を通過させてみよう

AEAD

AEAD(Authenticated Encryption with Associated Data)とは共通鍵暗号による秘匿性とMACによる完全性を両立した認証付き暗号のこと

TLS1.3ではAES-GCMやChaCha20-Poly1305が規格化されている



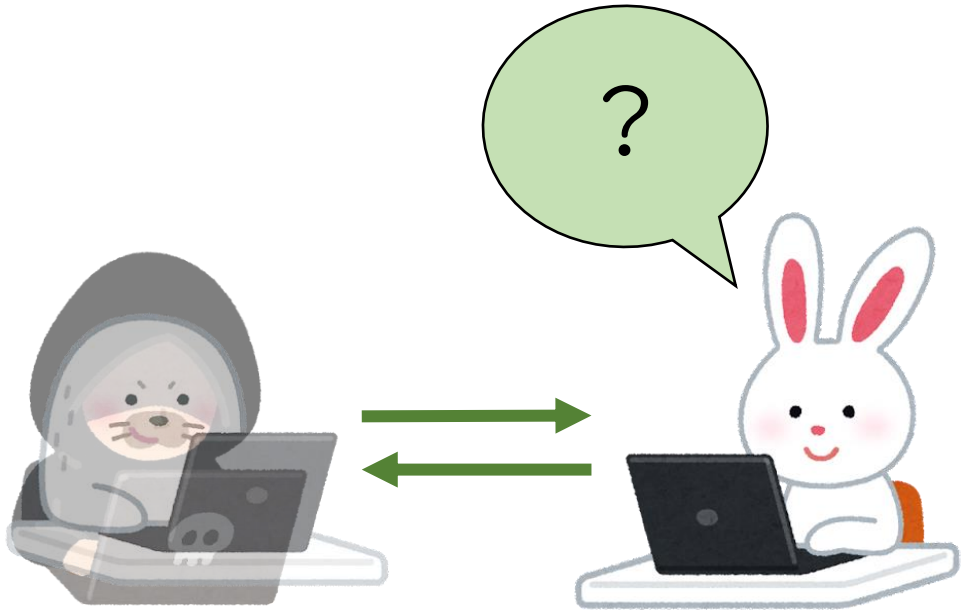
AEAD（演習）

- `6_message_authentication`ディレクトリにあるファイルを実行してみよう
- メッセージを途中で改ざんしたらErrorが返ってくることを確かめてみよう

デジタル署名

AEADの場合経路上でメッセージが改ざん検知による完全性と、暗号化による秘匿性を提供している

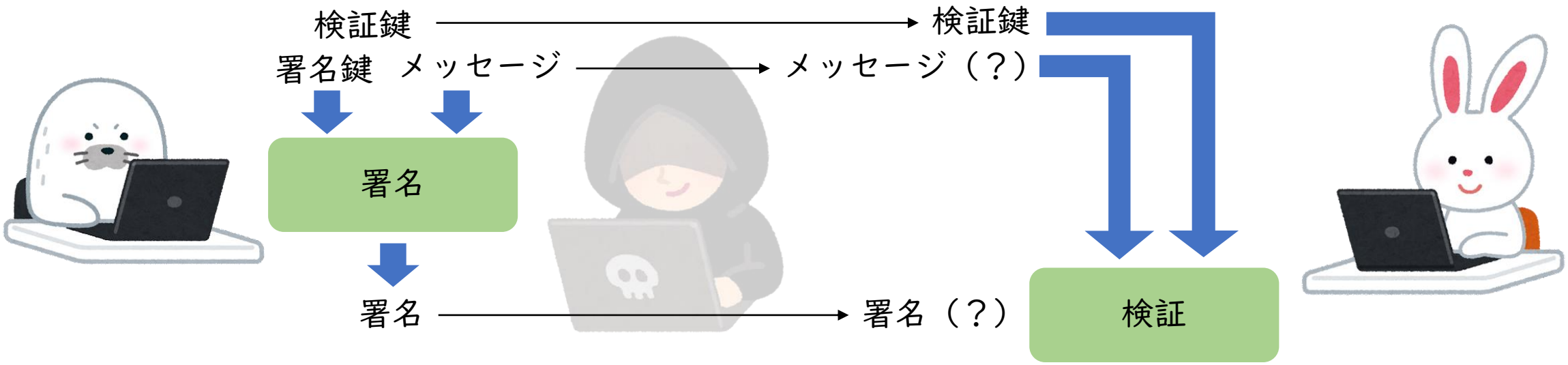
- 通信相手（あざらし）がほんとうにメッセージのやり取りをしたい本人かどうか分からない
- あざらしは自分にしかできない署名（Signature）を行うことで自分が送信した内容だと証明することができる



デジタル署名

現実世界では手書きのサインや捺印で署名を行うが、デジタルの世界では簡単にコピペできてしまう

デジタル署名ではメッセージに対して署名鍵（秘密鍵）で署名を行い、検証鍵（公開鍵）で検証を行う

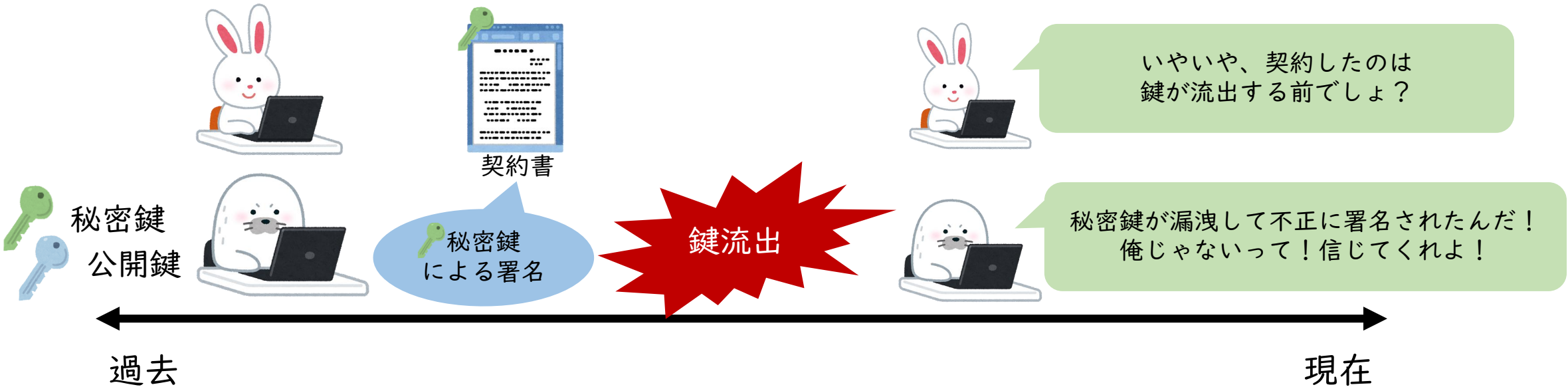


デジタル署名（演習）

- `openssl`コマンドを使用して`pubkey.pem`, `privkey.pem`を配置してみよう
- `7_digital_signature`にあるファイルを実行してみよう
- `digital_signature2.py`において署名鍵、検証鍵はなんという変数名か

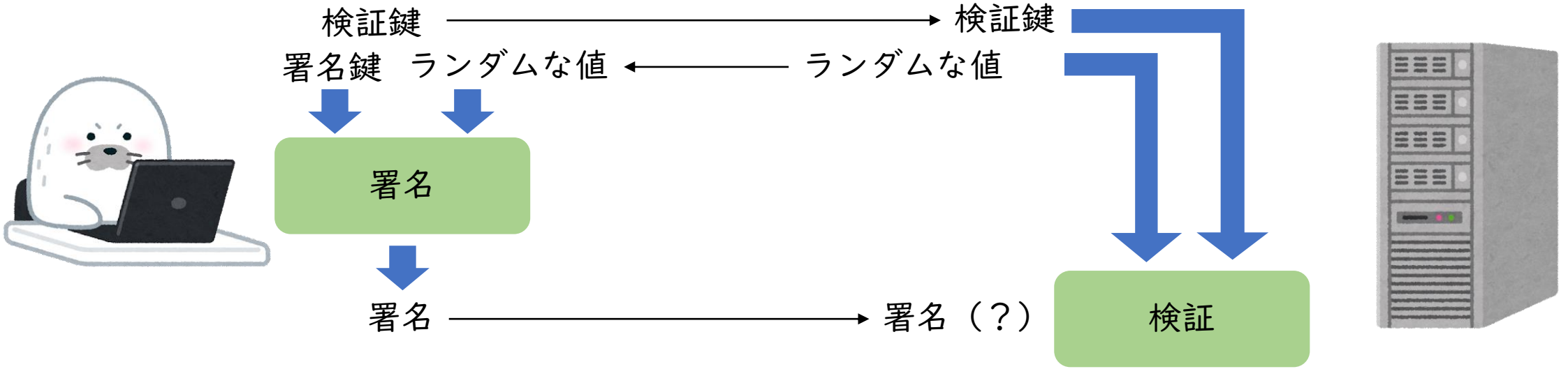
否認防止

否認防止とは契約書などに本人が署名を行い時間が経過してもその契約が言い逃れできないものにすることである。デジタル署名ではタイムスタンプと併用することで否認防止を実現している



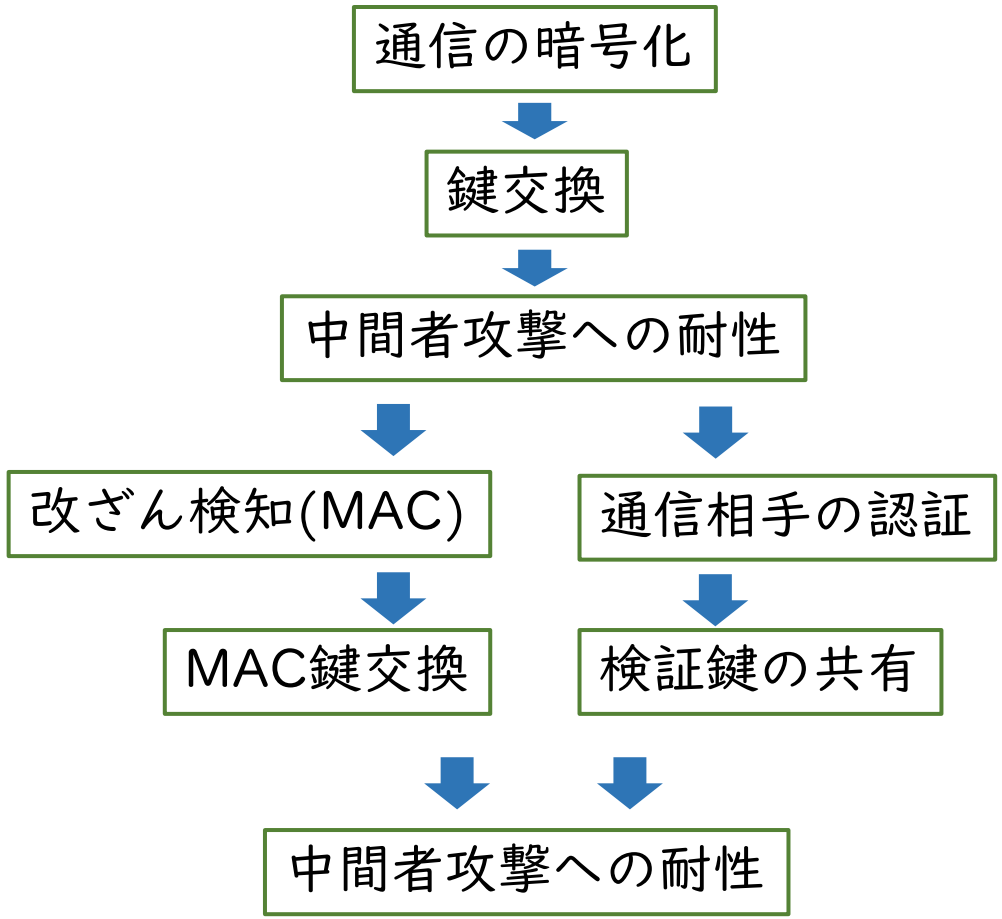
公開鍵認証

サーバにリモートでアクセスしたいときのSSHやGitHubで自分のリモートリポジトリにpushしたいとき、公開鍵認証を使用する

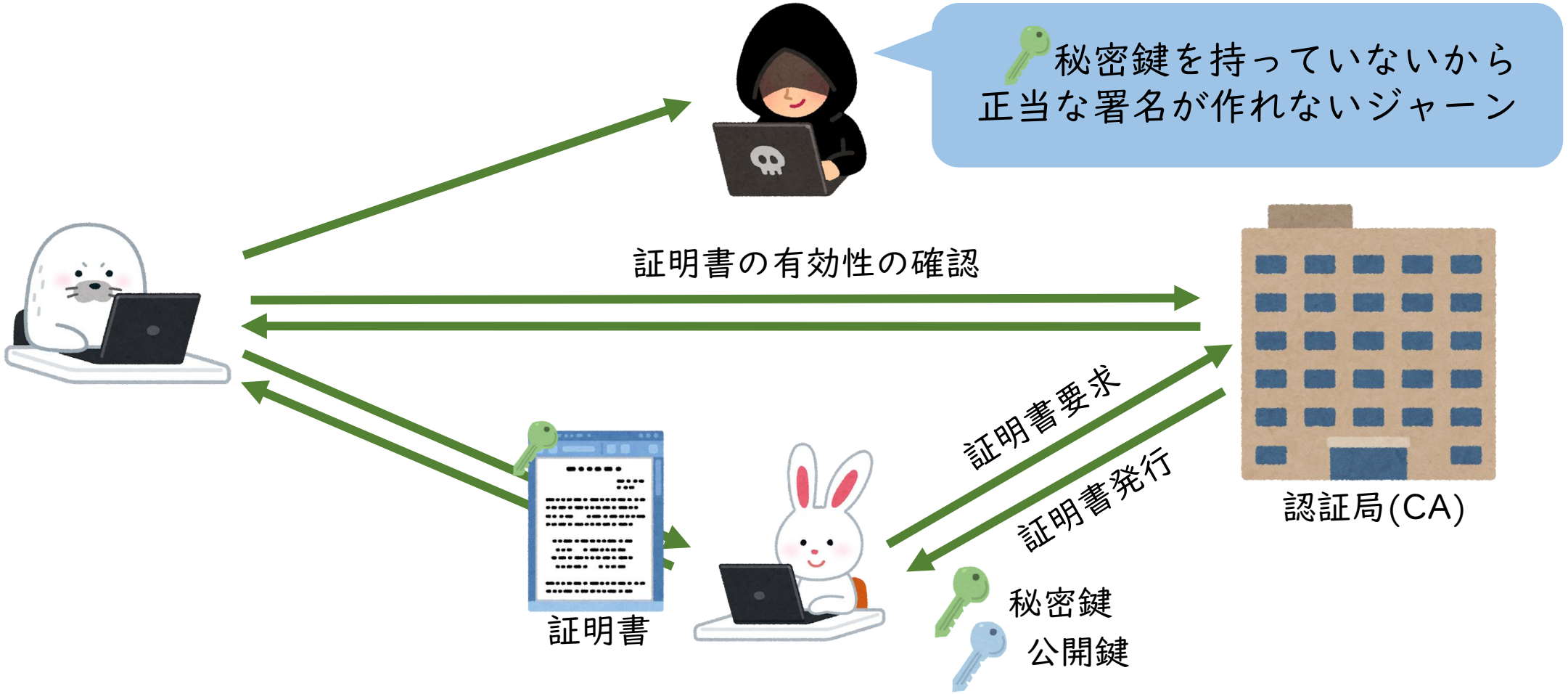


公開鍵基盤

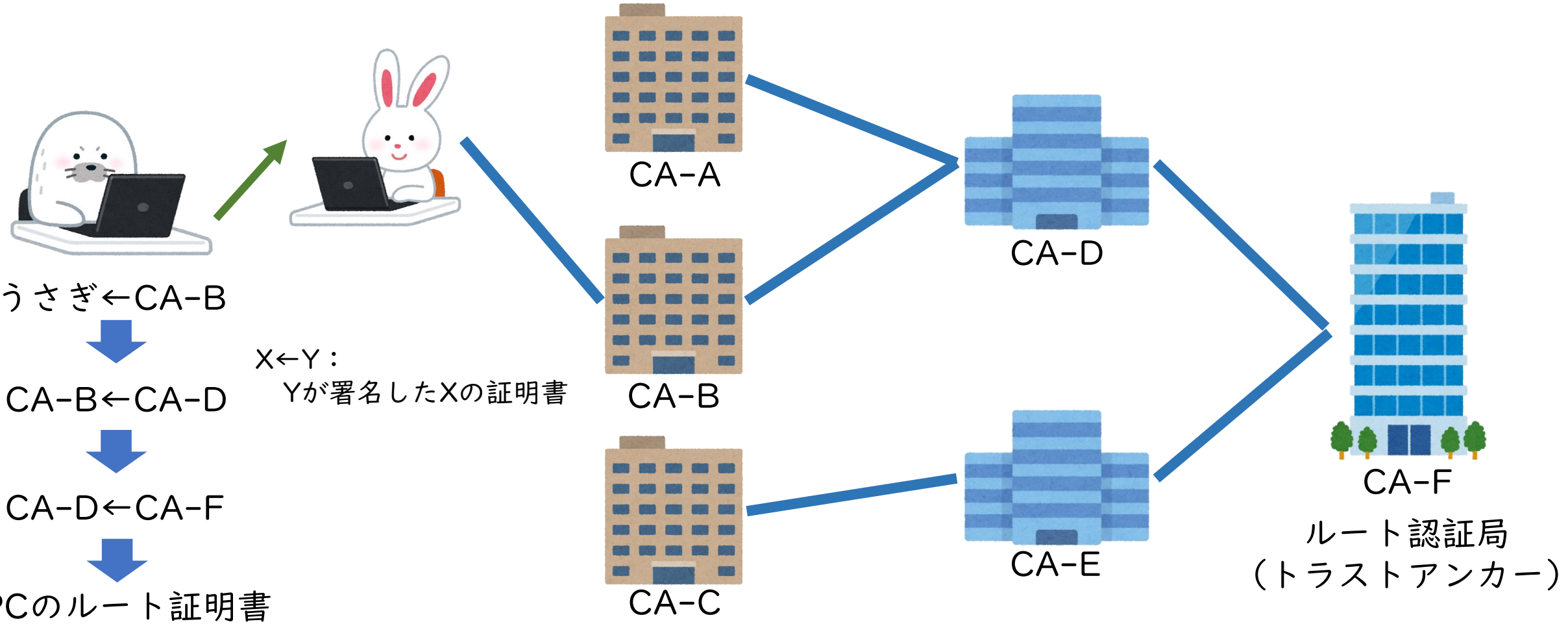
- 今までの暗号技術ではお互いの依存を断ち切ることができない
- 公開鍵とその持ち主を保証する仕組み（公開鍵基盤 PKI）が必要
- PKIの実現には証明書とそれを発行する認証局などの仕組みが使用されている



認証局による通信相手の認証



認証局と階層モデル


うさぎ←CA-B
CA-B←CA-D
CA-D←CA-F
PCのルート証明書
X←Y：
Yが署名したXの証明書
CA-A
CA-B
CA-C
CA-D
CA-E
CA-F
ルート認証局
（トラストアンカー）

証明書の発行

証明書要求者

公開鍵

秘密鍵

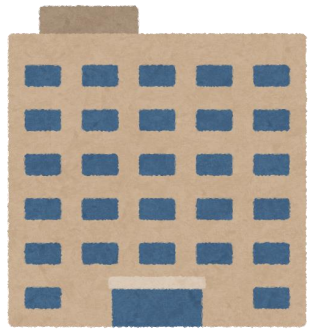
証明書要求
(CSR)

署名

署名

公開鍵

秘密鍵



認証局

証明書

証明書

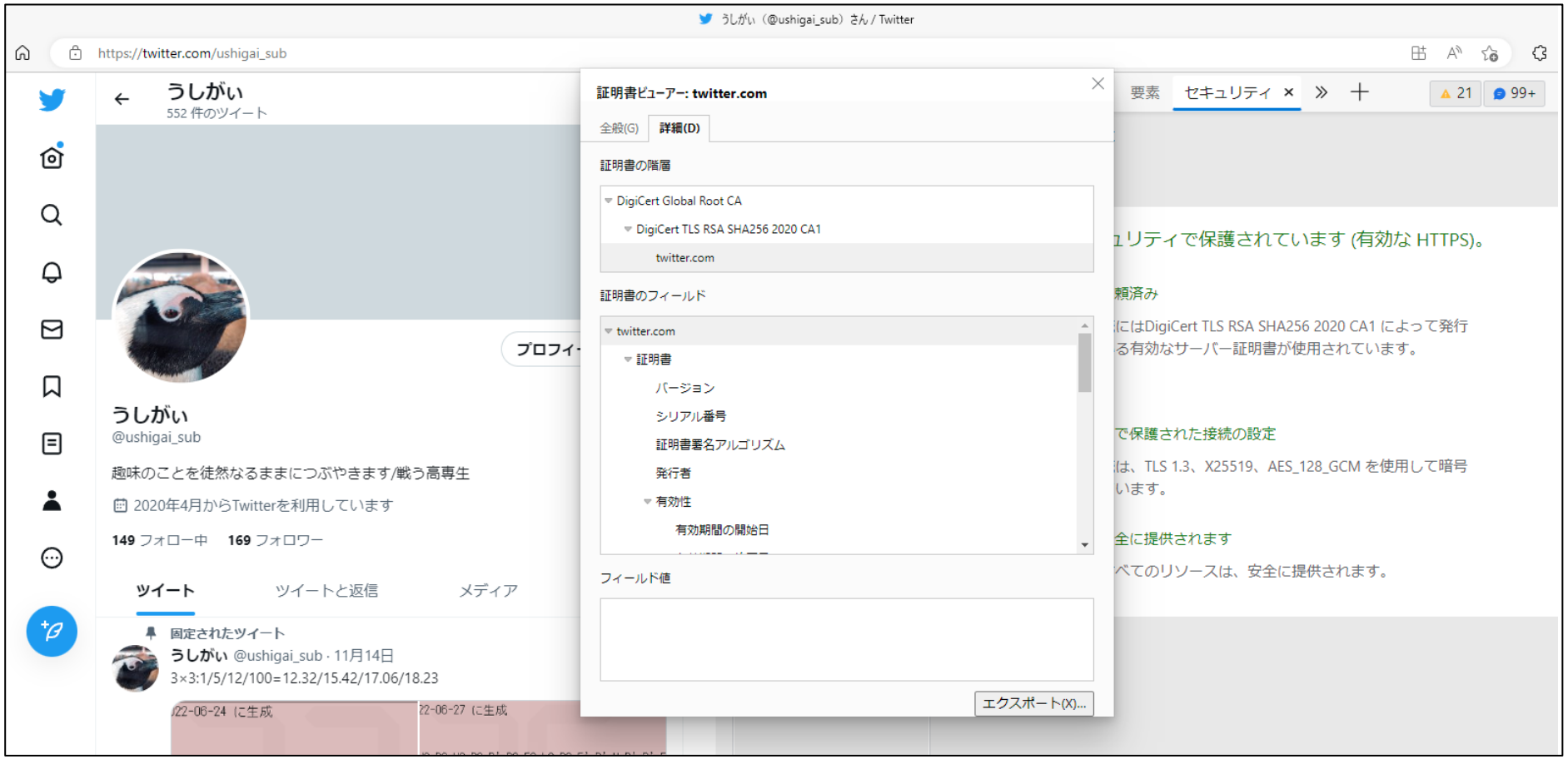
公開鍵証明書

- 公開鍵証明書が証明するもの
  - 所有者の公開鍵と秘密鍵
  - **所有者**情報
- 1996年X.509 v3証明書という規格が標準化され今日も広く使用されている

X.509 v3証明書の構造

バージョン
シリアル番号
署名アルゴリズム
発行者
有効期限
主体者（CAなど）
主体者公開鍵情報
一意の識別子
拡張

実際の証明書を確認してみる



実際の証明書を確認してみる

```
$ openssl genrsa 2048 > private.key # 2048bitのRSA秘密鍵の作成
$ cat private.key
$ openssl req -new -key private.key > server.csr # 証明書署名要求(CSR)の作成
$ cat server.csr
$ openssl genrsa 2048 > ca-private.key # CAの秘密鍵を作成
$ cat server.csr | openssl x509 -req -days 365 -signkey ca-private.key > server.crt # CSRをもとにCAが秘密鍵をもとに証明書(CRT)を作成
$ cat server.crt
$ cat server.crt | openssl x509 -text -noout # 証明書の内容を確認
```

演習

- 現実での身分証は顔写真や住所によって一意性を確保しているが、電子証明書の場合なにと紐づけることによってその一意性を確保しているか
- 証明書の秘密鍵が流出してしまったら？

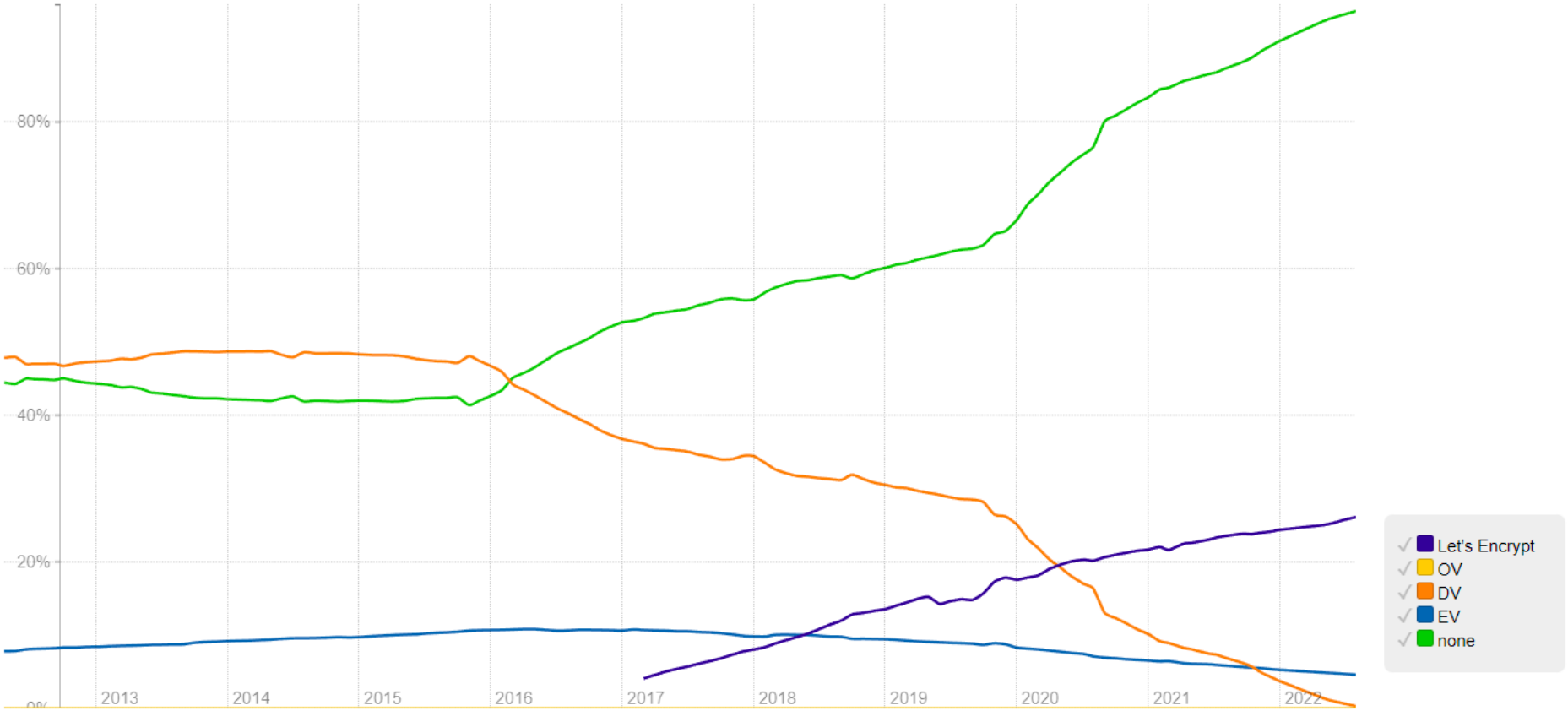
DV, OV, EV証明書

現実の身分証では顔写真や住所で一異性を確保している
デジタルの世界ではドメインや管理者の住所で保証している

種類	認証項目
DV証明書	申請者がドメインの利用権を持っているか
OV証明書	DV証明書での認証項目 申請者(組織)の法人番号確認
EV証明書	OV証明書での認証項目 申請者(組織)の住所 組織の運用状況

SSL Pulseによる統計

kjur.github.ioより引用

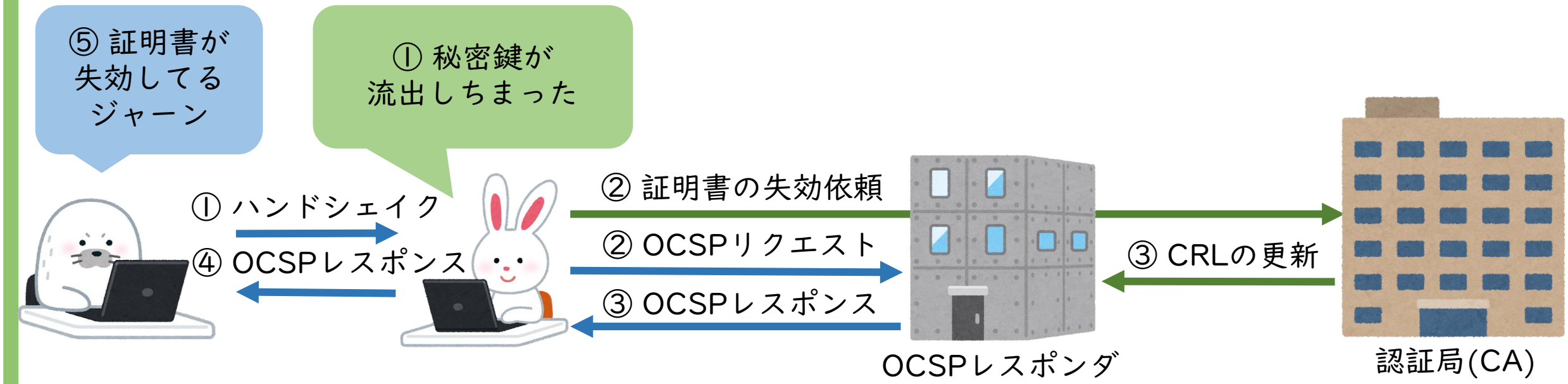


証明書の秘密鍵が漏洩してしまったら？

- 悪意のある攻撃者が秘密鍵で署名を偽造することができる
- 認証局に報告し署名を失効してもらう必要がある

OCSP Stapling

OCSP(Online Certificate Status Protocol) Staplingとはユーザが証明書の失効を確認するための仕組みである。OCSPレスポンダで証明書失効リスト(CRL)を保持しておき、クライアントがサーバにアクセスしたとき（ハンドシェイク時）にサーバがOCSPレスポンダに問い合わせてクライアントにOCSPレスポンスを返す。

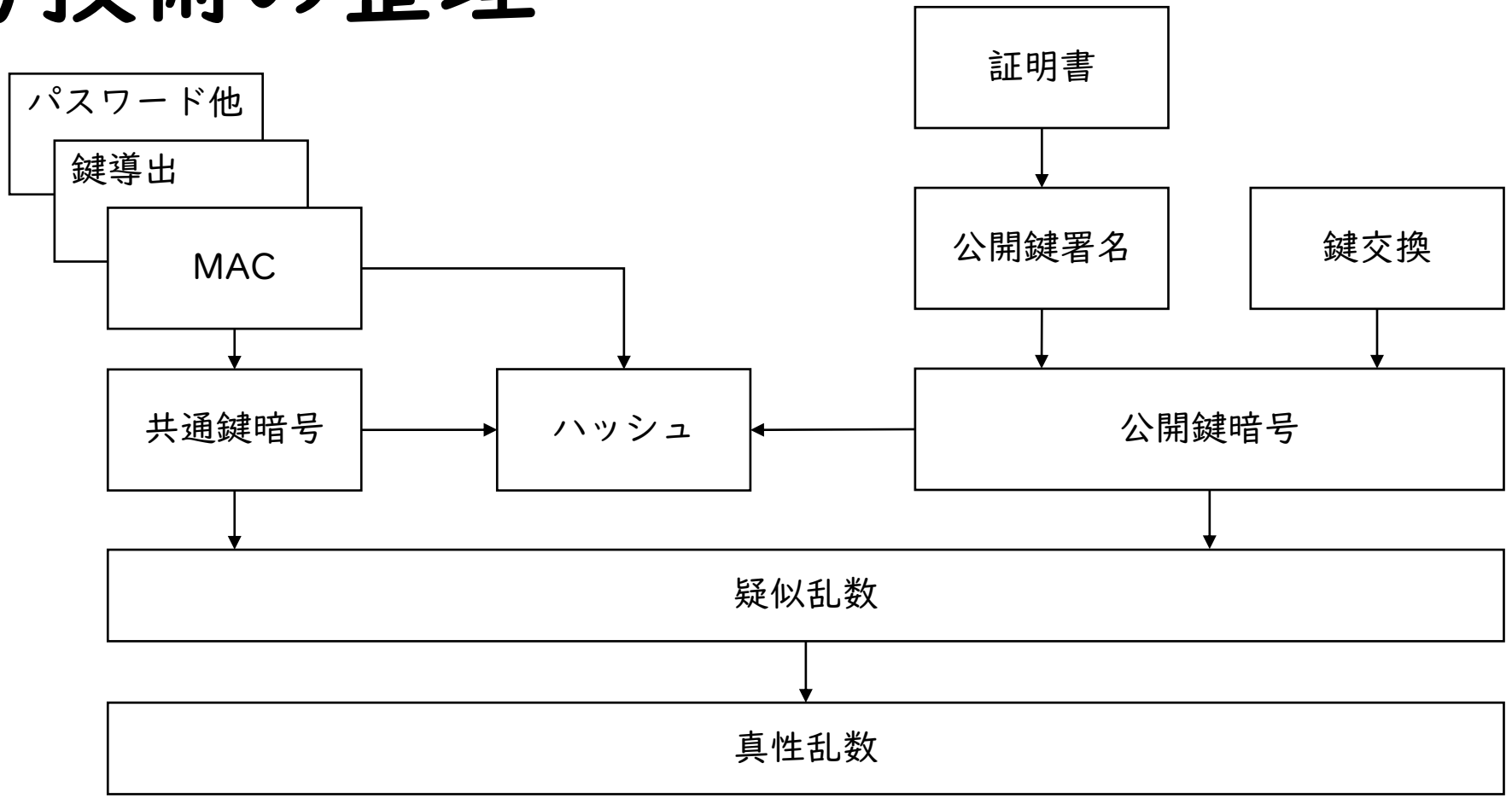


ブラウザの拡張機能による証明書検証

認証局が取り組む以外にもWebブラウザ内で証明書の検証を行っている場合がある

Webブラウザ	提供する機能
Google Chrome	CRLSets
Moizilla FIrefox	OneCRL
Microsoft Edge	Microsoft Defender SmartScreen

暗号技術の整理



参考『徹底解説TLS1.3（翔泳社）』p42 図3.1

ハンドシェイク(TLS1.2)

1. Client Hello

2. Server Hello

3. Key Exchange

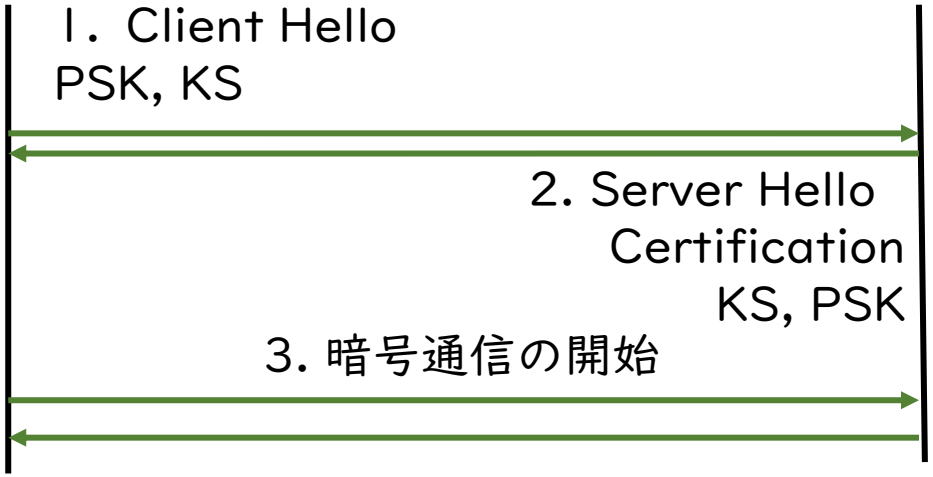
4. 暗号通信の開始

プリマスターシークレット

マスターシークレット

セッション鍵,IV,HMAC鍵

ハンドシェイク(TLS1.3)



TLS1.2と比較して暗号通信の開始までのプロセスが短くなってて、
暗号化して送信されるデータ（証明書など）も増えてるね

実際に使用されている暗号

私は…
鍵交換でECDHE
暗号化でAES-GCM
署名でRSAを使用します

私は…
鍵交換で静的RSA
暗号化でAES-CBC
署名でDSAを使用します

鍵長は増やせば増やすほど
安全って、ばあちゃんが
言ってた
AES-GCM1048576bit

→ SSL/TLSであらかじめ暗号アルゴリズムの組み合わせが決められており、
その組み合わせのことを（暗号スイート）という

暗号スイート（再掲）

opensslコマンドによる暗号スイートの確認方法を以下に示す
Transport Layer Security (TLS) Parameters (iana.org)も参照されたい

```
$ openssl ciphers -v
TLS_AES_256_GCM_SHA384 TLSv1.3 Kx=any Au=any Enc=AESGCM(256) Mac=AEAD
TLS_CHACHA20_POLY1305_SHA256 TLSv1.3 Kx=any Au=any Enc=CHACHA20/POLY1305(256)
Mac=AEAD
TLS_AES_128_GCM_SHA256 TLSv1.3 Kx=any Au=any Enc=AESGCM(128) Mac=AEAD
ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH Au=RSA Enc=AESGCM(256) Mac=AEAD
DHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=DH Au=RSA Enc=AESGCM(256) Mac=AEAD
ECDHE-ECDSA-CHACHA20-POLY1305 TLSv1.2 Kx=ECDH Au=ECDSA
Enc=CHACHA20/POLY1305(256) Mac=AEAD

  ~ snip ~
```

暗号スイート

TLS暗号設定ガイドライン
p19, 表6 暗号スイートでの利用推奨暗号アルゴリズムより引用

	CRYPTREC 暗号リストでの標記			備考
	技術分類	リストの種類	アルゴリズム名	
鍵交換	鍵共有・守秘	電子政府推奨暗号リスト	DHE (Ephemeral DH)	PFS 特性をもつ DHE を選択するのがセキュリティ上望ましい
			ECDHE (Ephemeral ECDH)	PFS 特性をもつ ECDHE を選択するほうがセキュリティ上望ましい
署名	署名	電子政府推奨暗号リスト	ECDSA	
			RSASSA PKCS#1 v1.5 (RSA)	
			RSASSA-PSS	TLS1.3 のみ利用可能
暗号化	128 ビットブロック暗号	電子政府推奨暗号リスト	AES	
			Camellia	TLS1.2 までで利用可能
	暗号利用モード	電子政府推奨暗号リスト	CCM	
			CCM_8	CCM の退縮版なので、CCM を利用するほうがセキュリティ上望ましい
			GCM	
	認証暗号	推奨候補暗号リスト	ChaCha20-Poly1305	
ハッシュ関数	ハッシュ関数	電子政府推奨暗号リスト	SHA-256	
			SHA-384	

利用推奨アルゴリズム

TLS暗号設定ガイドライン p42, 表17 推奨セキュリティ型での利用禁止暗号アルゴリズム一覧より引用

鍵交換		ECDHE, DHE
署名		ECDSA, RSASSA PKCS#1 v1.5 (RSA), RSASSA-PSS（TLS1.3 のみ）
暗号化	ブロック暗号	AES, Camellia（TLS1.2 のみ）
	暗号利用モード	GCM, CCM, CCM_8, CBC
	ストリーム暗号	ChaCha20-Poly1305
ハッシュ関数		SHA-256, SHA-384, SHA-1*)

利用禁止アルゴリズム

TLS暗号設定ガイドライン p42, 表16 推奨セキュリティ型での利用推奨暗号アルゴリズム一覧より引用

鍵交換		DH, ECDH
署名		GOST R 34.10-2012
暗号化	ブロック暗号	RC2, EXPORT-RC2, IDEA, DES, EXPORT-DES, GOST 28147-89, Magma, 3-key Triple DES, Kuznyechik, ARIA, SEED
	暗号利用モード	CTR_OMAC
	ストリーム暗号	RC4, EXPORT-RC4
ハッシュ関数		MD5, GOST R 34.11-2012

RFCによる規定

発行日	RFC番号	概要
03/2011	6176	SSL2.0の禁止
06/2015	7568	SSL3.0の廃止
06/2021	8996	TLS1.0,TLS1.1の廃止
02/2015	7465	RC4暗号スイートの廃止

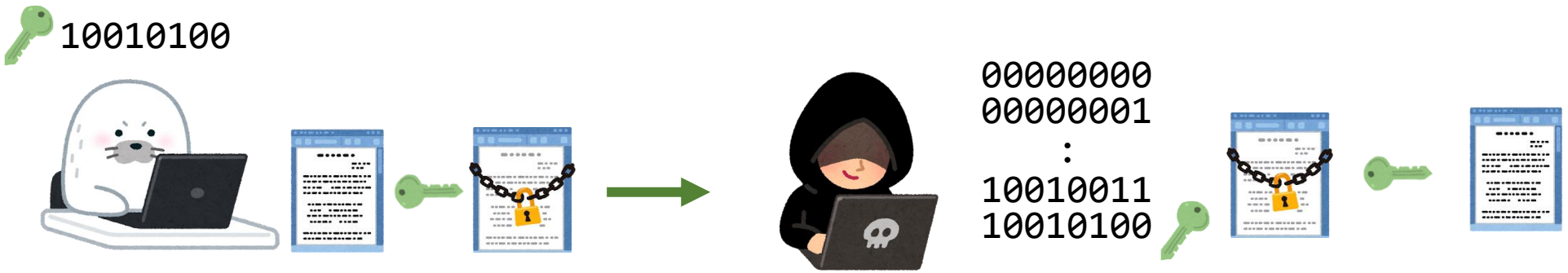
SSL Pulseでの統計

SSL Pulseによる主要15万サイトのプロトコルのサポート状況

	TLS1.3	TLS1.2	TLS1.1	TLS1.0	SSL3.0	SSL2.0
12/2022	58.9%	99.9%	37.0%	34.0%	2.1%	0.2%
12/2020	41.3%	99.1%	56.9%	50.1%	4.0%	0.6%
12/2018	10.5%	94.3%	79.1%	71.3%	8.7%	2.2%
12/2016	-	82.6%	80.0%	95.6%	19.2%	6.1%
12/2014	-	50.1%	47.4%	99.6%	53.5%	15.5%
12/2012	-	8.4%	6.7%	99.4%	99.8%	29.2%

暗号の安全性の指標

- 現実的な時間で解けるものと解けないものの境界はどこか
- 安全性の指標としてビットセキュリティが使われている
- 計算量が2^nのときその暗号をnビットセキュリティと呼ぶ



暗号の安全性の指標

暗号強度要件設定基準 (cryptrec.go.jp) 表5セキュリティ強度要件の基本設定方針より引用

想定運用終了・廃棄年／利用期間		2022～2030	2031～2040	2041～2050	2051～2060	2061～2070
112 ビット セキュリティ	新規生成 ((a)参照)	移行完遂期間 ((c)参照)	利用不可	利用不可	利用不可	利用不可
	処理 ((b)参照)		許容			
128 ビット セキュリティ	新規生成 ((a)参照)	利用可	利用可	移行完遂期間 ((c)参照)	利用不可	利用不可
	処理 ((b)参照)				許容	
192 ビット セキュリティ	新規生成 ((a)参照)	利用可	利用可	利用可	利用可	利用可
	処理 ((b)参照)					
256 ビット セキュリティ	新規生成 ((a)参照)	利用可	利用可	利用可	利用可	利用可
	処理 ((b)参照)					

SSL/TLSでの暗号とビットセキュリティ

公開鍵暗号の推定セキュリティ強度

セキュリティ強度（ビットセキュリティ）	IFC RSA-PSS RSASSA-PKCS1-v1.5 RSA-OAEP RSAES-PKCS1-v1_5	FFC DSA DH	ECC ECDSA ECDH PSEC-KEM
112	k = 2048	(L, N) = (2048, 224)	P-224 B-233 K-233
128	k = 3072	(L, N) = (3072, 256)	P-256 B-283 K-283 W-25519 Curve25519 Edwards25519
192	k = 7680	(L, N) = (7680, 384)	P-384 B-409 K-409 W-448 Curve448 Edwards448
256	k = 15360	(L, N) = (15360, 512)	P-521 B-571 K-571

[cryptrec 暗号強度要件（アルゴリズム及び鍵長選択）に関する設定基準](#)
p8表2より引用

鍵長のトレードオフ性

RSAに156bitのビットセキュリティを持たせるには鍵長を15360bitにする必要がある。57680bitの素数を生成するために必要な時間は？
暗号を運用するうえでは適切な鍵長を設定する必要がある

```
>>> from Crypto.Util.number import *
>>> from time import time
>>>
>>> def f(bits):
...     start = time()
...     getPrime(bits)
...     print(time()-start)
...
>>> f(3840)
39.238982915878296
```

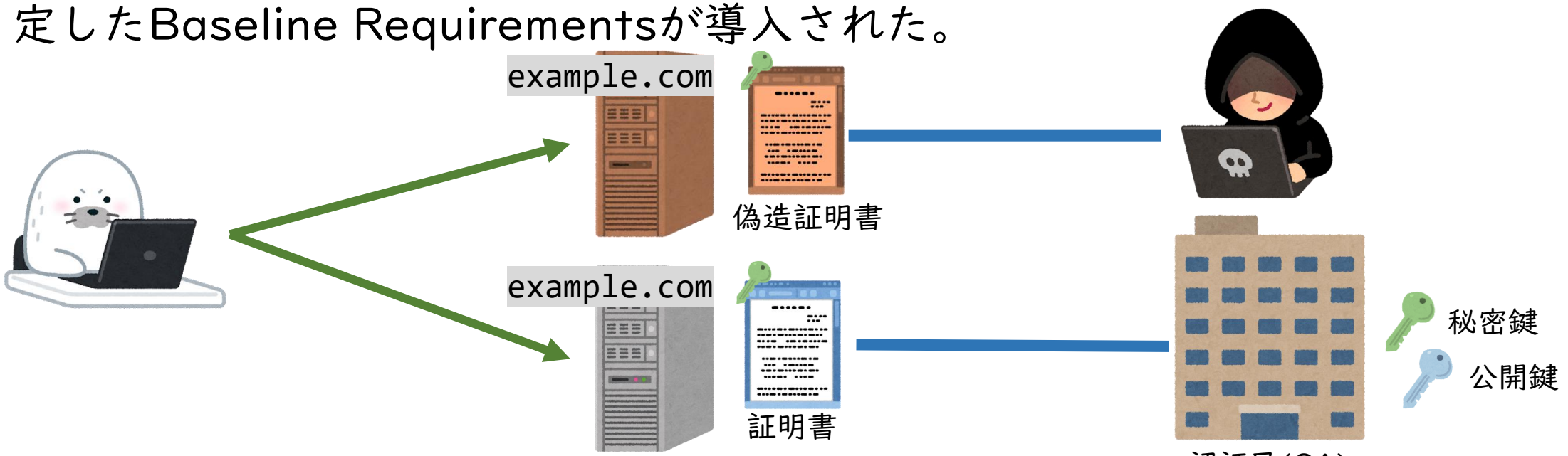
SSL/TLS通信に存在する脅威

SSL/TLSは通信をサーバ,クライアント間の通信を保護し、機密性,完全性,真正性を保護する。しかし暗号アルゴリズムに対してピンポイントに脆弱性が見つかることはまれでも、実装上の問題やサーバの設定不備など特定の条件下で問題を引き起こす可能性が存在する。

- Heartbleed Bug
- 暗号の危殆化
- 量子コンピュータの登場
- 認証局への侵害
- フィッシングサイトのSSL/TLS化
- CookieのSecure属性

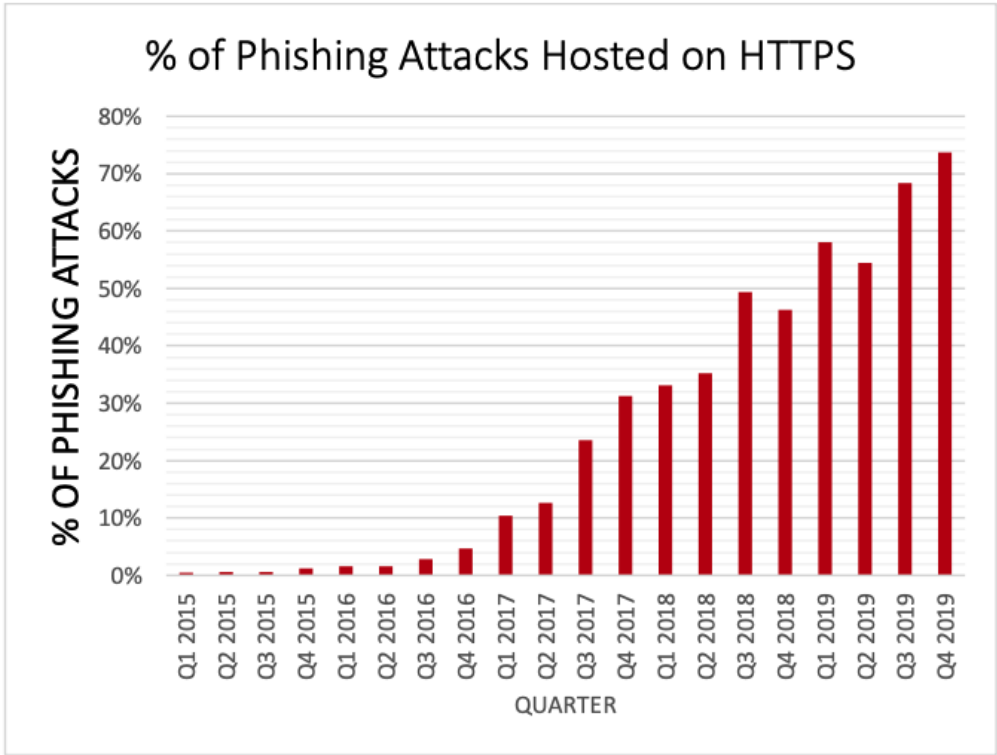
認証局への侵害

- 2011年オランダのルート認証局DigiNotarが侵害され少なくとも531枚の偽造証明書が発行された
- この事件以降ドメイン管理者が自身に対して発行される証明書を任意に許可しないCAAや、従業員の管理体制を把握し運用状況のチェックを規定したBaseline Requirementsが導入された。



フィッシングサイトのSSL/TLS化

- 近年HTTPS化が推進されWebブラウザではHTTPSでないものは「保護されていないサイト」など警告を出すようにしている
- しかしフィッシングサイトでは2019年の時点で70%以上がHTTPS化しているという統計がある
- フィッシングサイトへの誘導はSMSがよく使用されているのでユーザー側の注意が必要

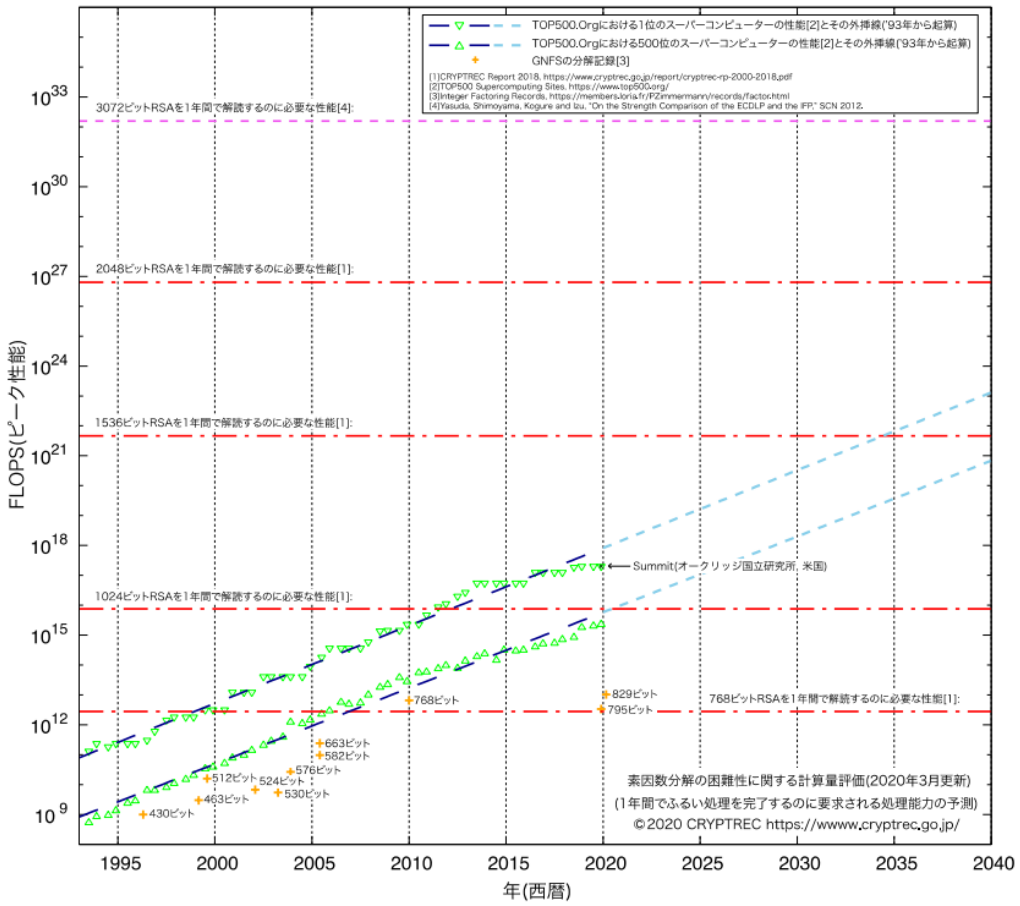


Heartbleed Bug

- OpenSSLの拡張である「Heartbeat」にあるプログラムのミスによるバグ
- Heartbeat要求のサイズと応答のサイズがチェックされないこと利用して1度に最大64KBのデータ（e.g. 利用者のセッションキー, サーバの秘密鍵）をメモリから抜き出すことができる
- OpenSSLはApacheやnginxでも使用されていたので影響は広かった

暗号の危殆化

- 112bit以上のビットセキュリティを持つ暗号は2030年でも安全とされている
- ある日突然画期的なアルゴリズムが発見されて少ない計算量で解読されてしまう可能性がある
- 日本の場合CRYPTRECが運用監視暗号リストを公開しているので監視が必要である



素因数分解の困難性に関する計算量評価

CRYPTREC暗号技術検討会２０２１年度 報告書図3.2-1より引用

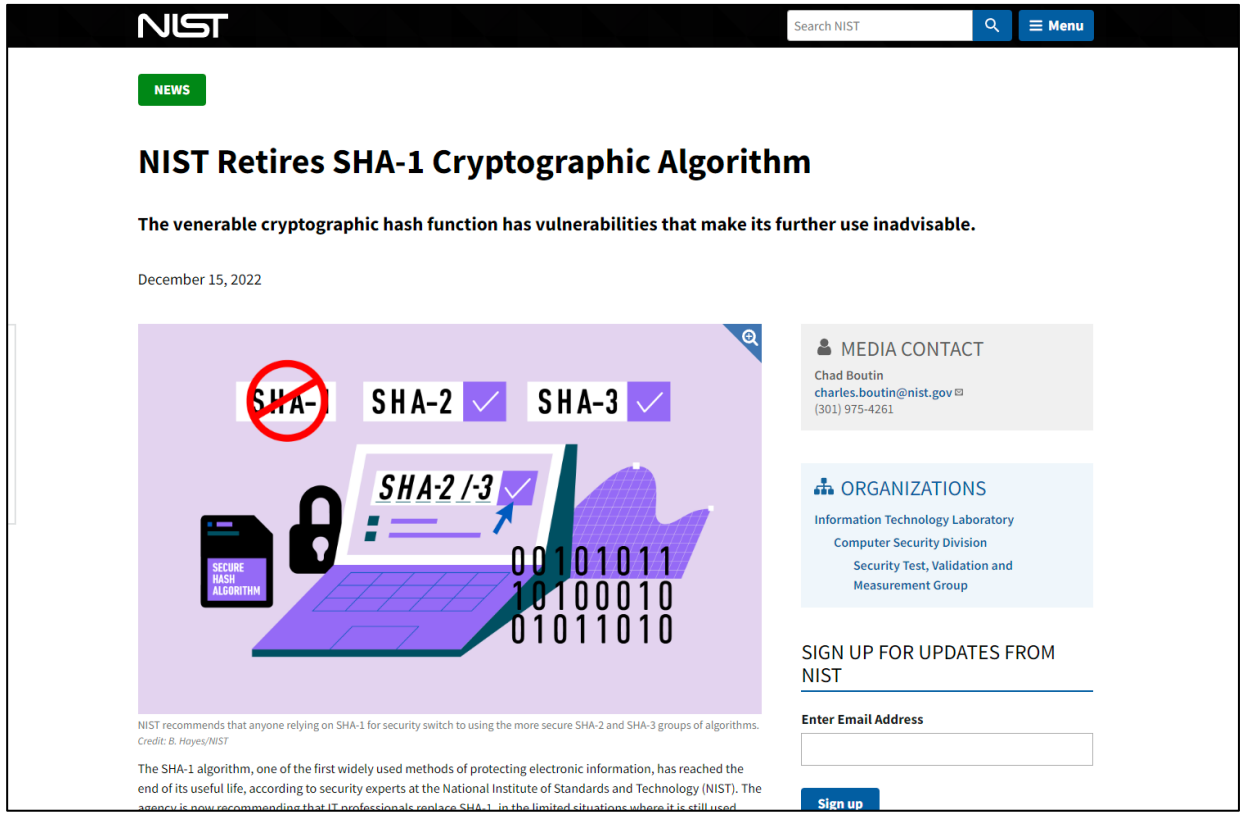
量子コンピュータの登場

現在私たちが使用するコンピュータは0,1という2つの状態をもつ。量子コンピュータは0の状態と1の状態を重ね合わせて表現されている。

- 共通鍵暗号に対する影響
  - AESにおいて現行利用されている暗号利用モードで、鍵長の半分程度のビットセキュリティに低下させる攻撃手法がある
- 公開鍵暗号に対する影響
  - ShorのアルゴリズムをRSA4096bitでも36bitのビットセキュリティ程度の攻撃可能である
  - ただしCRYPTRECの調査によると15の素因数分解に失敗したという発表があり、また4096bitの素因数分解には最低8192bitの量子ビットが必要である

最近の動向 (1/3)

NIST Retires SHA-1 Cryptographic Algorithm



NIST

Search NIST

Menu

NEWS

NIST Retires SHA-1 Cryptographic Algorithm

The venerable cryptographic hash function has vulnerabilities that make its further use inadvisable.

December 15, 2022

NIST recommends that anyone relying on SHA-1 for security switch to using the more secure SHA-2 and SHA-3 groups of algorithms.
Credit: B. Hayes/NIST

The SHA-1 algorithm, one of the first widely used methods of protecting electronic information, has reached the end of its useful life, according to security experts at the National Institute of Standards and Technology (NIST). The agency is now recommending that IT professionals replace SHA-1, in the limited situations where it is still used

MEDIA CONTACT
Chad Boutin
charles.boutin@nist.gov
(301) 975-4261

ORGANIZATIONS
Information Technology Laboratory
Computer Security Division
Security Test, Validation and Measurement Group

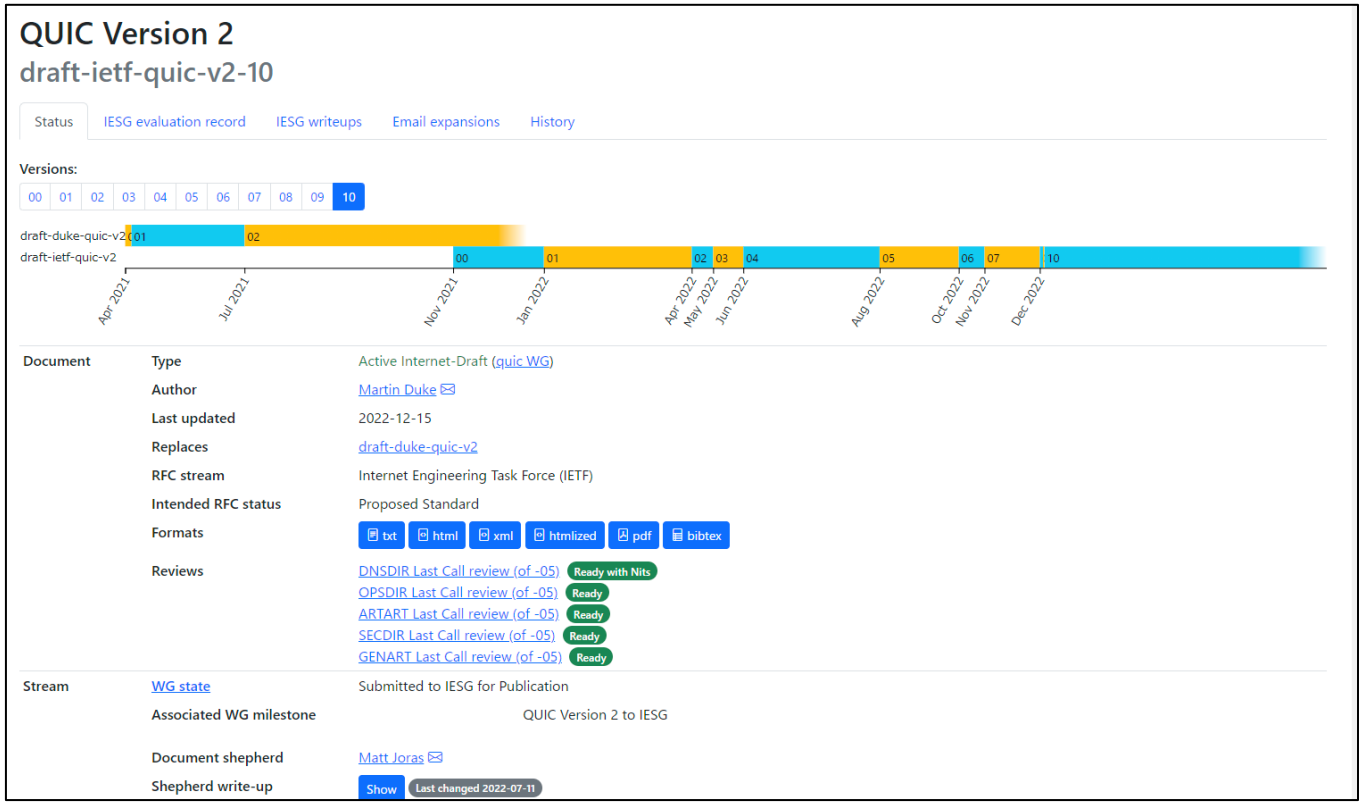
SIGN UP FOR UPDATES FROM NIST

Enter Email Address

Sign up

最近の動向 (2/3)

draft-ietf-quic-v2-10 - QUIC Version 2



最近の動向 (3/3)

https://pc.watch.impress.co.jp/docs/news/1463929.html

まとめ

- SSL/TLSはセキュア通信プロトコルと呼ばれており、機密性,完全性,真正性を保ちたい
- 機密性を保つには共通鍵暗号による暗号化を、完全性を保つにはHMACによる改ざん検知を、真正性を保つには証明書と署名検証アルゴリズムがある
- ハンドシェイクで通信相手の認証と鍵共有を行い、暗号通信では暗号化とHMACを両立したAEADによるメッセージのやり取りを行っている

参考 (1/4)

- Wikipedia
  - X.509, 暗号論的疑似乱数生成器, 公開鍵基盤, ハッシュ関数, Advanced Encryption Standard, 暗号利用モード, Transport Layer Security
- IETFよりRFC関連, https://www.ietf.org/rfc/
  - RFC 8446 – The Transport Layer Security (TLS) Protocol Version 1.3
  - RFC 5246 – The Transport Layer Security (TLS) Protocol Version 1.2
  - RFC 5705 – Keying Material Exporters for Transport Layer Security (TLS)
  - RFC 5280 – Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
  - RFC 6234 – US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)
  - RFC 7568 – Deprecating Secure Sockets Layer Version 3.0
  - RFC 6176 – Prohibiting Secure Sockets Layer (SSL) Version 2.0
  - RFC 2104 – HMAC

参考 (2/4)

- NIST SP800 https://csrc.nist.gov/publications/sp800
  - SP800-57 Recommendation for Key Management
  - SP800-131A Transitioning the Use of Cryptographic Algorithms and Key Lengths
  - SP800-90A Recommendation for Random Number Generation Using Deterministic Random Bit Generators
  - SP800-22 A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications
- Jean-Philippe Aumasson,『暗号技術 実践活用ガイド』, マイナビ出版, 2020年
- 古城 隆ほか,『徹底解剖 TLS1.3』, 翔泳社, 2022年
- 光成 滋生,『暗号と認証のしくみと理論がこれ1冊でしっかりわかる教科書』, 技術評論社, 2021年
- Joshua Davies, 『Implementing SSL/TLS』, Wiley, 2010年
- Neeru Mago PMAC: A Fully Parallelizable MAC Algorithm, https://acfa.apeejay.edu/docs/volumes/journal-2015/paper-05.pdf

参考 (3/4)

- TLS暗号設定ガイドライン, https://www.ipa.go.jp/security/ipg/documents/ipa-cryptrec-gl-3001-3.0.1.pdf
- APWG Year-End Report: 2019 A Roller Coaster Ride for Phishing, https://www.phishlabs.com/blog/top-phishing-trends-2019/
- The Heartbleed Bug, https://heartbleed.com/
- CRYPTREC 暗号技術検討会 2021年度 第1回, https://www.cryptrec.go.jp/report/cryptrec-rp-1000-2021.pdf
- CRYPTREC 量子コンピュータ時代に向けた暗号の在り方, https://www.cryptrec.go.jp/report/cryptrec-mt-1441-2020.pdf
- CRYPTREC 暗号強度要件（アルゴリズム及び鍵長選択）に関する設定基準,https://www.cryptrec.go.jp/list/cryptrec-ls-0003-2022.pdf
- IPA 暗号鍵管理システム 設計指針, https://www.ipa.go.jp/security/ipg/documents/ipa-cryptrec-gl-3002-1.0.pdf

参考 (4/4)

- SSL Pulse, https://www.ssllabs.com/ssl-pulse/
- Symantec 認証局におけるハッキング事件の原因と認証局業界の新しい取り組み, https://www.digicert.co.jp/welcome/pdf/wp_cahacking.pdf
- Symantec SSL ハンドシェイクの裏側, https://www.digicert.co.jp/welcome/pdf/wp_ssl_handshake.pdf
- herumi, RSA署名を正しく理解する, https://zenn.dev/herumi/articles/rsa-signature
- 0a24, TLS1.3の中身をみよう（RFC8448）, https://zenn.dev/0a24/articles/tls1_3-rfc8448
- wolfSSL, 真性乱数生成器 VS 疑似乱数生成器, https://wolfssl.jp/wolfblog/2021/08/16/true-random-vs-pseudorandom-number-generation/
- The Heartbleed Bug, https://heartbleed.com/
- ISMS 情報セキュリティマネジメントシステムとは, https://isms.jp/isms/index.html