End-to-End セキュリティを担保する 3
セキュリティエンジニアリング特論 第 5 回

栗原 淳

兵庫県立大学大学院

2022-11-03

はじめに

はじめに

前２回で，

- End-to-End (E2E) セキュリティの原則と必要性
- Web サイトでの E2E セキュリティ実践のため，標準化技術を用いた共通鍵暗号の利用方法

学んだ．

今回は「公開鍵暗号」の利用方法を見ていく．

公開鍵暗号って？

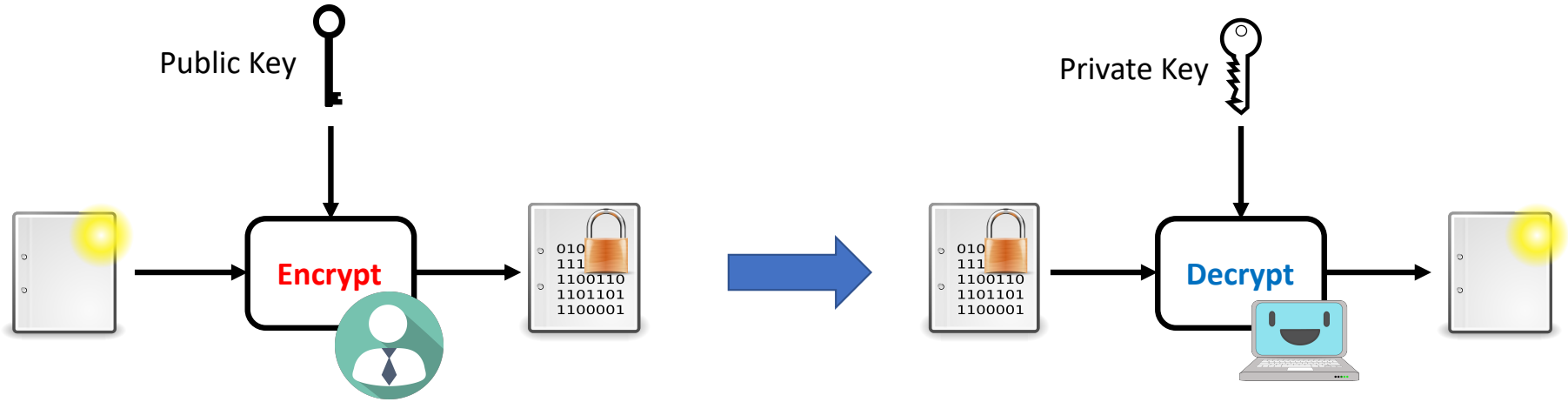
既知だと思うが，まずざっと定義しておく．

定義: 公開鍵暗号

以下のステップで暗号化・復号が行われる暗号方式のこと

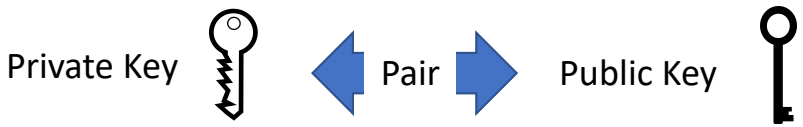
1. 特殊な数学的条件を満たす鍵ペア「公開鍵 PK と 秘密鍵 SK」を生成
2. PK は公開，SK は秘匿
3. データ D を PK によって暗号化して，暗号化データ X を生成
4. X は SK によってデータ D に復号される．

暗号化・復号の鍵を分けて，暗号化の鍵を公開してしまうことでパスワードなどの共有が不要になる．



Anyone can encrypt information only with public key without knowing private key!

Only the private key owner can decrypt the information encrypted under its paired public key.



⇒ 非常に強力な暗号化の概念．現代のセキュリティインフラはこれで成り立っていると言っても過言ではない．

今回は，システムにおいて正しく・安全に公開鍵暗号を使っていくためのお話．

今回の講義で最終的に学びたいこと

- 公開鍵暗号はどういうものか．共通鍵暗号と比べた pros/cons．
- RSA 暗号と楕円曲線暗号[1] の違いと，その標準化手法．
- 共通鍵と公開鍵暗号を組み合わせてデータを暗号化するために．

[1] 楕円曲線 Diffie-Hellman を取り上げる

公開鍵暗号の使い方 事始め

公開鍵暗号の種類

公開鍵暗号の定義「特殊な数学的条件を満たす鍵ペアを生成」

⇓

この「数学的条件」に複数の種類が存在．

JavaScript に限らず，各種環境で利用可能な代表的な公開鍵暗号：

- 素因数分解に関する条件
  → RSA 暗号
- 楕円曲線上の離散対数に関する条件
  → 楕円曲線暗号 (Elliptic Curve Cryptography)[2]

この２つの使い方，注意ポイントを今回は取り上げる．

[2]今回は便宜上 Elliptic Curve Diffie-Hellman; ECDH を楕円曲線暗号と呼んでいく．

RSA 暗号のさわり

RSA Cryptography

言わずもがな，公開鍵暗号の代表的な手法

- 1977 年，Rivest-Shamir-Edelman の 3 名により発明．2000 年に特許期間満了 (現在特許フリー)．暗号化以外に「署名」の手法への応用も有名．
- パディング方式まで含めた手法が，RFC 8017 (PKCS#1 v2.2)，ANSI X9.31，IEEE 1363，CRYPTREC 等，各所で標準に採用．
- 公開鍵長は 1024–4096bits が標準的に使われている．[3]
- 暗号化・署名の際には，元のデータにパディングが必要．パディング方法によりセキュリティが大きく左右される．[4]

[3] 原理的には無限に伸ばせる．

[4] RSA-OAEP(暗号化)，RSA-PSS(署名) が現状ベターな方法．これを話す．

楕円曲線暗号のさわり

Elliptic-Curve Cryptography

楕円曲線という数の世界での「離散対数」を使った方式の総称[5]

- 1985 年頃，Victor Miller，Neal Koblitz により独立に考案．
- Diffie-Hellman(DH)[6] を楕円曲線上で実行するのが ECDH，DSA[7] を楕円曲線上で実行するのが ECDSA．
- RFC8442，CRYPTREC，IEEE P1363 等で標準化．TLS や Bitcoin など多方面で利用．
- 公開鍵長は 256–521bits (Compact form) が標準的に使われている．
- ECDH は「ECDH-Ephemeral」という運用方法を用いることで，普通に使うより安全性が大きく向上する．[8]

[5] 普通に離散対数問題を使うより，楕円曲線上でやることで安全性を担保する鍵長が短くなる．

[6] RFC2631 https://tools.ietf.org/html/rfc2631

[7] NIST FIPS 186-4 https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[8] Forward Secrecy(後述) を担保する．

AES と比べた公開鍵暗号の Pros/Cons

何でもかんでも公開鍵暗号，で良さそうな気もしてくるが…

	Pros	Cons
AES	・安全性を担保する鍵長が短い (128bits〜) ・一般的に高速・SoC での最適化も望める[9]	・パスワードなどの事前共有が必要
公開鍵暗号	・パスワードなどの秘密情報の事前共有が不要	・安全性を担保する鍵長が長い (RSA: 2048bits〜) ・一般的に非常に遅い・重い

⇒ 使い所を考えて組み合わせて使う，もしくは場合に応じて使い分けないと実用に耐えないシステム・サービスが出来上がる.

[9] Intel AES-NI

安全性を担保する鍵長が大きく違うのはどういうこと？

AES と比べた RSA・楕円曲線暗号の公開鍵のビット長比較[10]．
横 1 行がだいたい同じくらいの安全性と言われる．

AES	RSA	楕円曲線
128	3072	256–383
192	7680	384–511
256	15360	512–

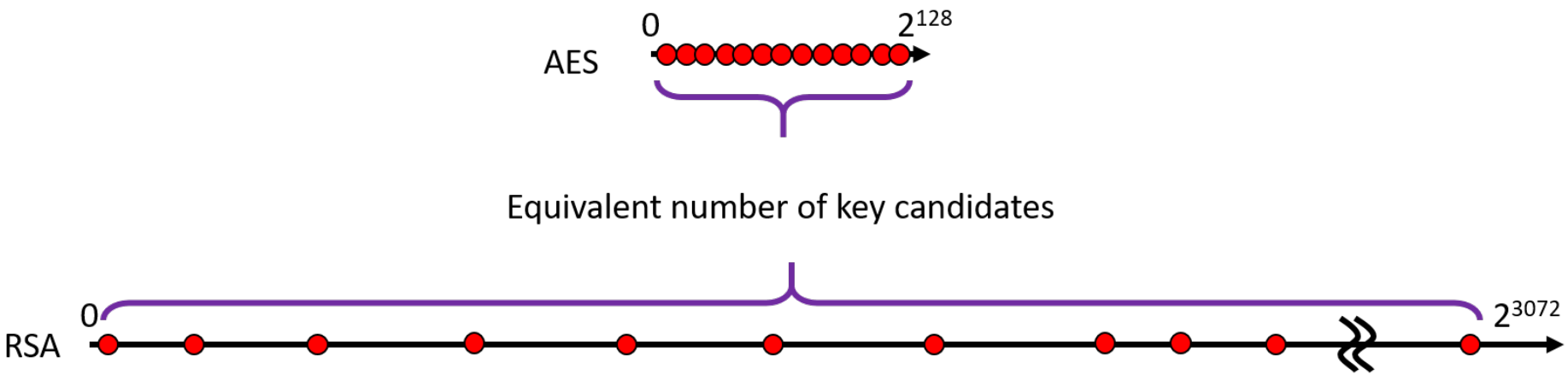
AES に比べて，楕円曲線で倍，RSA に至っては 24 倍以上の鍵長を使わないと，同じくらいの安全性を担保できない．

鍵長が長いほど，暗号化・復号がどんどん重く・遅くなっていく…

[10] Recommendation for Key Management, Special Publication 800-57 Part 1 Rev. 4, NIST, 01/2016. https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-4/final

「AES-128 が，RSA-3072 と同じくらい」というイメージは，以下のように説明できる．

- AES: 数値 $= 0, 1, \ldots, 2^{128} - 1$ のうち，どれか 1 つが鍵．
- RSA: 特殊な条件を満たす数 = 素数 2 個の積 (合成数) を選んで，公開・秘密鍵を求める．

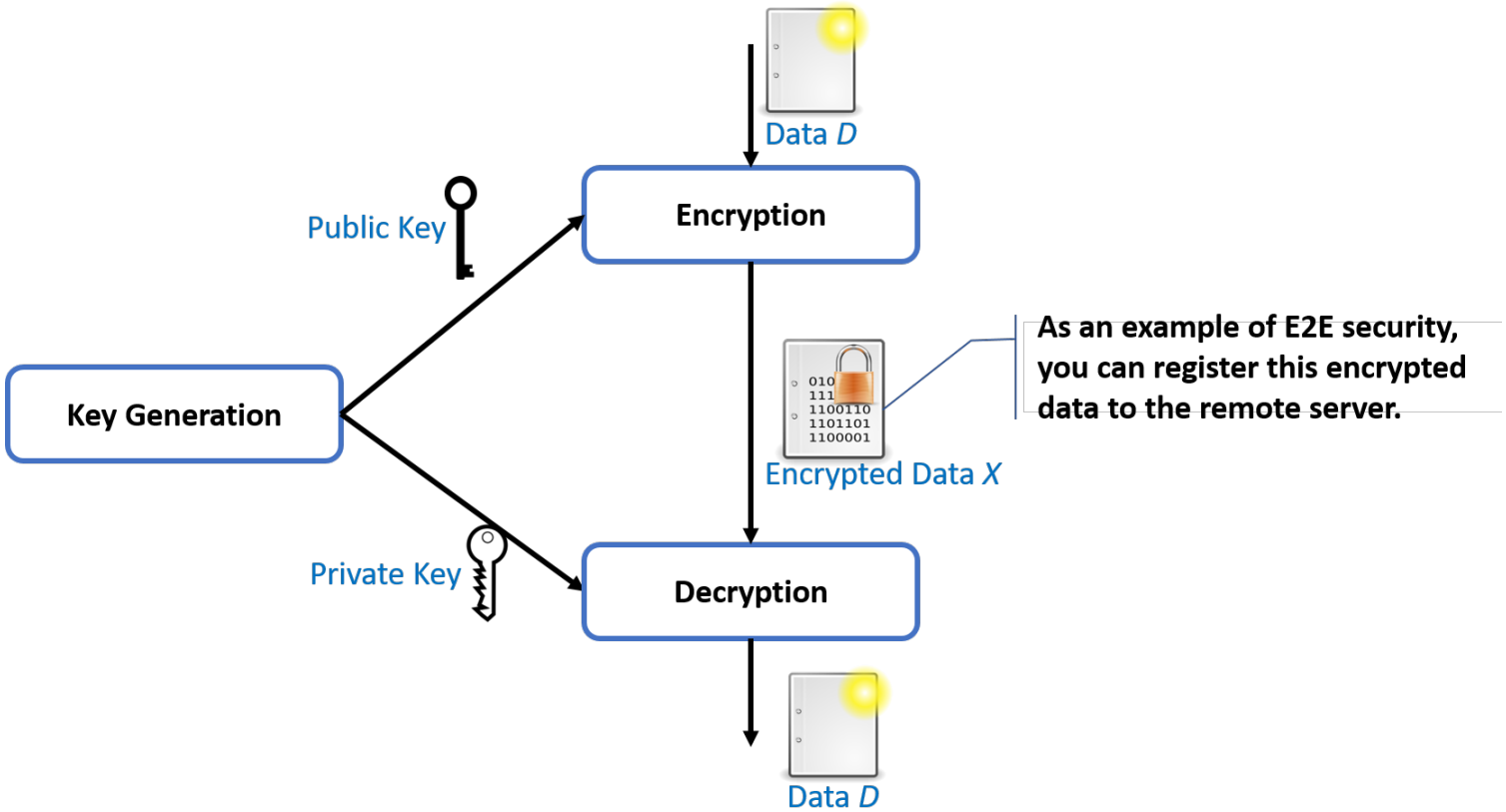


総当たりした時に「当たる」確率を揃えるには，RSA はその分巨大な数まで候補にしないとならない．

サンプルコードの準備

準備

細かく暗号化の説明を聞きつつ，手を動かすため，まず環境準備．
今回は，JavaScript (Node.js) を使って手元で公開鍵暗号化・復号．



サンプルコードはブラウザでも動く．src/commands-browser.html を開くとこれから Node.JS で試すデモが開発者コンソールで実行される．適宜試したり比較すると良い．

前回・前々回使った「リモートサーバに登録する」というところは，簡略化のため省略した．興味があれば，前回のコードを公開鍵暗号に拡張して，ネットワークを介して実験してみよう．

環境

以下の環境が前提:

- Node.js (≥ v18) がインストール済．yarn が使えること．[11]
- ブラウザとして，Google Chrome (系ブラウザ)，もしくは Firefox がインストール済み
- Visual Studio Code や WebStorm などの統合開発環境がセットアップ済みだとなお良い．

[11] インストールコマンド: `npm i -g yarn`

JavaScript プロジェクトの準備

- プロジェクトの GitHub リポジトリ[12]を Clone

```
$ git clone
https://github.com/junkurihara/lecture-security_engineering.git
$ cd sample-05
```

- 依存パッケージのインストール

```
$ yarn install
```

- ライブラリのビルド

```
$ yarn build
```

[12] https://github.com/junkurihara/lecture-security_engineering

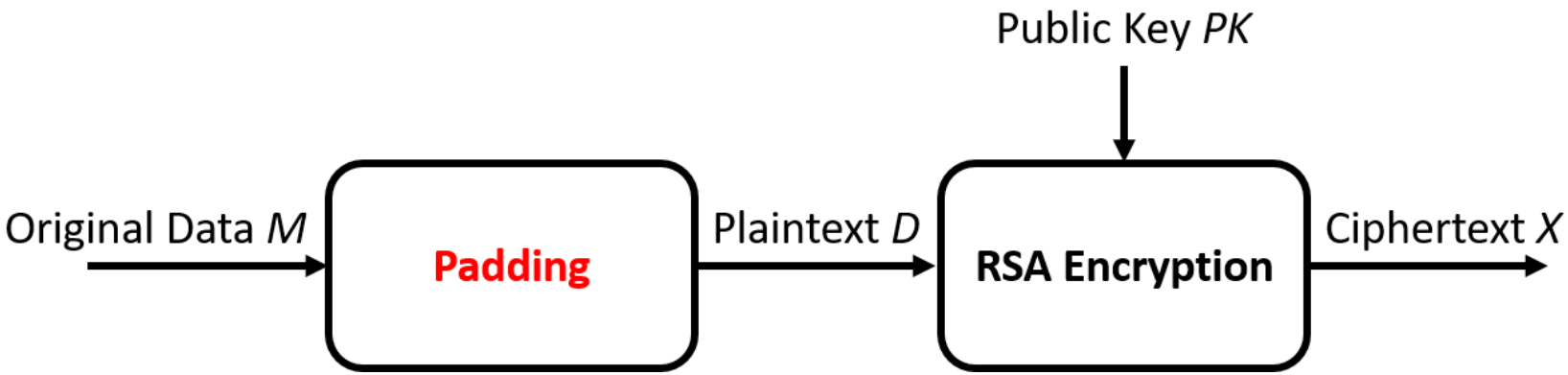
RSA 暗号

RSA 暗号を使うための作法

RSA 暗号化の制限

データ D と，公開鍵 PK とが，同じビット長でなければならない

⇒ RSA 暗号化の前には，まず元データへのパディング[13]が必要．



RSA 暗号化には，前処理のパディング方法の選択が最重要

[13] 長いデータの場合は切断…

RSA 向けに主として 2 種類のパディング方法が知られている．[14]

- PKCS#1-v1.5 Padding
- Optimal Asymmetric Encryption Padding (OAEP)

[14] 共に PKCS#1 (RFC8017) で標準化．https://tools.ietf.org/html/rfc8017

PKCS#1-v1.5 Padding

- RSA 暗号化まで入れて RSAES-PKCS1-v1_5 と呼ぶ [15].
- 元データ M に，公開鍵長まで以下のようなパディングを付与.

$$D = \texttt{0x00} \,||\, \texttt{0x02} \,||\, \mathrm{RandomSequence} \,||\, \texttt{0x00} \,||\, M$$

- 暗号化データを任意に改変でき，復号者に復号成功・失敗を確認させられる時，元データを復号される脆弱性が知られている. [16]
- PKCS#1 v2.2 （RFC8017) で「後方互換性のため以外では使用するな」と明示的に記載．CRYPTREC においても推奨暗号方式リストからドロップ. [17]

基本的に使うな

[15] RSA Encryption Scheme

[16] 1998 年の Bleinchenbacher's Attack． 2018 年，現代の Internet でも未対策ホスト・サービスが大量なことが発表されている (ROBOT Attack).

[17] `https://www.cryptrec.go.jp/method.html`

Optimal Asymmetric Encryption Padding (OAEP) [18]

- RSA 暗号化と組み合わせて，RSA-OAEP，もしくは RSAES-OAEP と呼ぶ．
- 元データ M とランダムシードに対して，All or Nothing Transform (AONT) を実行し，D を公開鍵ビット長まで膨らませる．

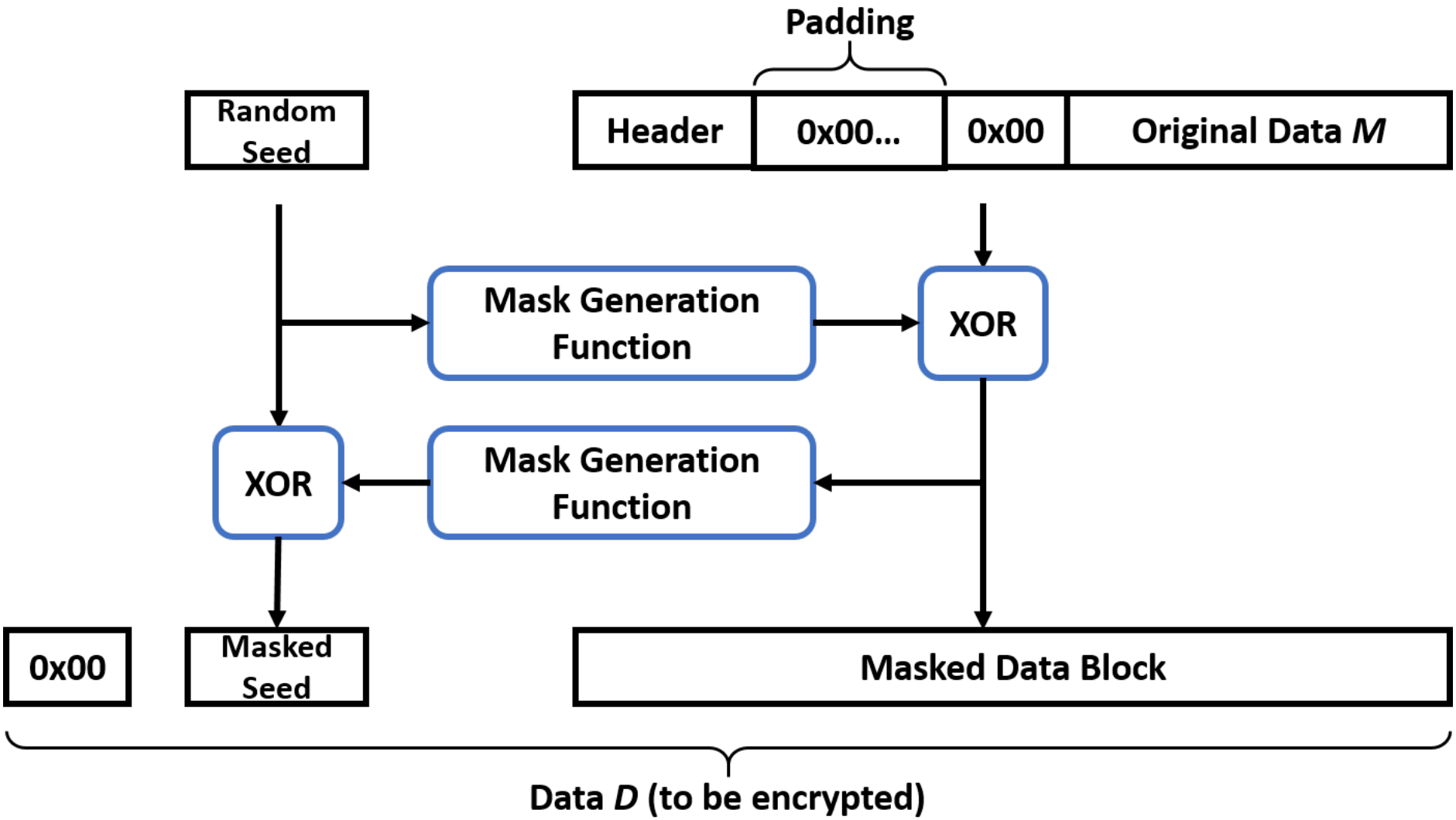
$$D = \mathrm{AONT}(M, \mathrm{RandomSeed})$$

- PKCS#1-v1.5 Padding の脆弱性は潰されている．実用の上ですぐに致命的な脆弱性は知られていない．PKCS#1 v2.2 (RFC8017) では，新規アプリは OAEP を利用することと明記．

今は RSA なら OAEP 使っとけば間違いない

[18] M. Bellare and P. Rogaway, “Optimal Asymmetric Encryption,” in Proc. EUROCRYPTO 1994, pp. 92–111, LNCS 950, 1994.

OAEP の処理フロー



逆変換は Masked Seed, Masked Data Block が両方揃えば可能.

JavaScript で RSAES-OAEP

sample ディレクトリで以下を実行すると，公開鍵・秘密鍵ペアを生成して，引数の string の暗号化→復号を一連で実行する．

```
$ yarn execute rsa-oaep-demo 'hello world'
<Input Data>
hello world
<Generated RSA Key Pair (PEM Form)>
Public Key:
30820122300d06092a864..... // 生成した公開鍵 (DER)
Private Key:
308204bc020100300d060..... // 生成した秘密鍵 (DER)
=======

<Encrypted Data (in Base64)>
9f28a2acbd7cd5bc748f3..... // 'hello world' の RSA-OAEP 暗号化データ
=======

<Decrypted Data>
hello world // 復号したデータ
=======
```

一連の動作をオートでやらずに自力でやる方法 (コピペで頑張る):

RSA 鍵ペアを生成

```
$ yarn execute rsa-keygen
<Generated RSA Key Pair (DER Form)>
Public Key:
30820122300d06092a864886f70d010......
Private Key:
308204be020100300d06092a864886f......
```

RSA-OAEP 暗号化 (-p で公開鍵を指定)

```
$ yarn execute rsa-oaep-encrypt 'hello world'\
 -p '308201223......'
<Encrypted Data (in HexString)>
8da122191b1ec6da72afe88c96cfbb3...... // 暗号化データ
```

RSA-OAEP 復号 (-s で秘密鍵を指定)

```
$ yarn execute rsa-oaep-decrypt '8da122191b1ec6da72afe88c96cfbb3......'\
 -s '308204be020100300d06092a864886f......'
<Decrypted Data>
hello world
```

RSA-OAEP による公開鍵暗号化のコードはこんな感じ．

RSA 鍵ペア生成 (src/test-api.js)

```
// bits = 2048
const jscu = getJscu(); // jscu オブジェクト取得．Node.js Crypto，WebCrypto のラッパー
const keyPair = await jscu.pkc.generateKey(
  'RSA',
  {modulusLength: bits} // 2048bits の RSA 鍵ペアを生成
);
```

RSA-OAEP 暗号化 (src/test-api.js)

```
const jscu = getJscu(); // jscu オブジェクト取得．Node.js Crypto，WebCrypto のラッパー

// DER エンコード (Uint8Array) の公開鍵を jscu の鍵オブジェクトに変換．
const publicKey = new jscu.Key('der', publicDer);

const encrypted = await jscu.pkc.encrypt(
  uint8ArrayData, // 暗号化されるデータ
  publicKey,
  {hash: 'SHA-256'} // OAEP で利用されるハッシュ．今なら'SHA-256' 使っとけばいい．
);
```

RSA-OAEP 復号 (src/test-api.js)

```
const jscu = getJscu(); // jscu オブジェクト取得. Node.js Crypto, WebCrypto のラッパー

// DER エンコード (Uint8Array) の秘密鍵を jscu の鍵オブジェクトに変換.
const privateKey = new jscu.Key('der', privateDer);

const decrypted = await jscu.pkc.decrypt(
  uint8ArrayEncryptedData,
  privateKey,
  {hash: 'SHA-256'} // 暗号化で使われているものと一緒.
);
```

今回も栗原のライブラリを使ってネイティブ API はラッピングしている.

RSAES-OAEP は，WebCrypto API(ブラウザ) でも，Node.js Crypto でもネイティブサポートされている．[19]

なお，RSAES-PKCS1-v1_5 も規格上サポートされている．JS に限らず，どちらかというと OAEP の方がまだ実装されていない．

⇒ そのせいで，SSL/TLS では ROBOT 攻撃に対する脆弱性とかを引き起こしている……

[19] ただし，全てのブラウザでサポートされているわけではない (WebCrypto)．栗原ライブラリは purejs で実装し直しているので動かないことはない．IE と Edge は今すぐ捨てよう．

楕円曲線暗号 (ECDH)

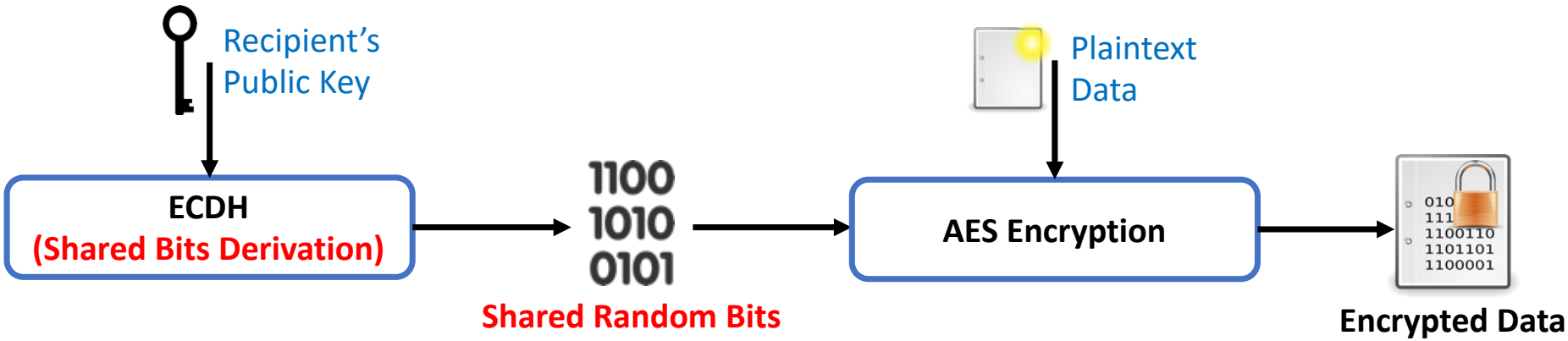
楕円曲線暗号を使うための作法 その 1

ECDH は厳密には「公開鍵暗号」ではない

Elliptic-Curve Diffie-Hellman (ECDH)

ECDH 自身は，データの暗号化ではなく，公開鍵・秘密鍵を使って送受信者間で秘密裏にランダムビット列を共有するための方法．

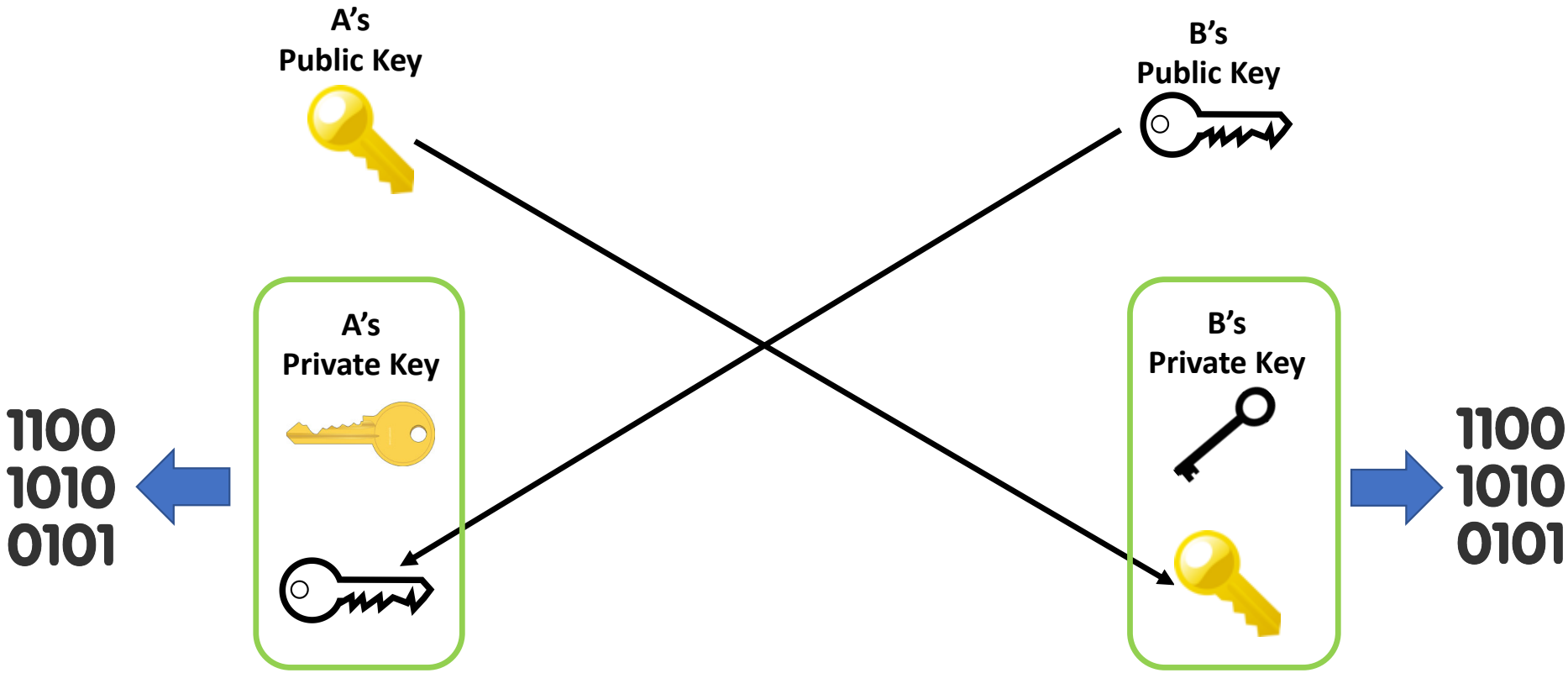
⇒ 共有したランダムビット列を鍵 (の種) として用いて，AES とかでデータを暗号化．



この流れ全体で「公開鍵暗号」の体を為す．

というわけで，まずはこの「共有ランダムビット列」の導出の話．

(EC)DH における共有ランダムビット列の導出の流れ:



Just by simply exchange the public key each other, one can calculate the shared random bits from the other's public key and its own private key.

送信者 (A) と受信者 (B) が，互いに公開鍵を交換するだけで同じビット列を秘密裏[20]に共有できる．

[20]秘密鍵は一切表に出てこないことに注目

ちょっとフォーマルに書くと:

ECDH の特徴 [21]

2 つの公開鍵・秘密鍵ペア: (PK_1, SK_1) と (PK_2, SK_2) で，

$$\begin{aligned}\texttt{SharedRandomBits} &= \mathrm{ECDH}(PK_1, SK_2) \\ &= \mathrm{ECDH}(PK_2, SK_1).\end{aligned}$$

すなわち ECDH では，鍵ペアを持っているもの同士なら，相手の公開鍵から共有ビット列が導出可能.

共有ビット列の長さは，公開鍵長 (Compact form) と一緒.

[21] EC でない DH も同様.

JavaScript で ECDH 共有ビット列導出をしてみよう

sample ディレクトリで以下を実行．鍵ペアを 2 つ生成→互いの公開鍵から共有ビット列を生成する．

```
$ yarn execute check-ecdh
<ECC Key Pair A (DER Form)>
Public Key:
3059301306072a8648ce3d020106082... // 公開鍵 A
Private Key:
308193020100301306072a8648ce3d0... // 秘密鍵 A
=======
<ECC Key Pair B (DER Form)>
Public Key:
3059301306072a8648ce3d020106082... // 公開鍵 B
Private Key:
308193020100301306072a8648ce3d0... // 秘密鍵 B
=======

// 公開鍵 A と秘密鍵 B から生成
Shared Bits from Public Key A and Private Key B: c55393fc681811141...
// 公開鍵 B と秘密鍵 A から生成
Shared Bits from Public Key B and Private Key A: c55393fc681811141...
```

共有ビット列は全く同じものになっていることに注目

ECDH 共有ビット列導出のコードの中身はこんな感じ．

src/test-api.js

```
const jscu = getJscu();
const jscec = getJscec(); //js-crypto-ec オブジェクト．jscu のサブモジュール．

// DER エンコードから jscu 鍵オブジェクトを生成，そのあと JWK エンコードの鍵として出力
const publicKey = new jscu.Key('der', publicDer);
const privateKey = new jscu.Key('der', privateDer);
const publicJwk = await publicKey.export('jwk');
const privateJwk = await privateKey.export('jwk');

// 共有ビット列の出力
const derived = await jscec.deriveSecret(publicJwk, privateJwk);
```

※ ECDH の仕組みをわかりやすくするためにサブモジュールを呼んでいる．

ECDH の共有ビット列導出は，WebCrypto API(ブラウザ) でも，Node.js Crypto でもネイティブサポートされている．[22]

[22] ブラウザでは encrypt などというわかりやすい API ではなく，deriveBits(ビット導出) という API．Node.js では ECDH オブジェクト生成．

楕円曲線暗号を使うための作法 その2

前回の学んだことの復習.

前回の復習

マスターシークレット (=ECDH 共有ビット列) を元に AES 暗号化を行うためには，マスターシークレットから作る鍵のランダム性の向上，鍵の総当たりの困難化を施す.

⇒ 共有ビット列に対して鍵導出関数を利用することで担保する.

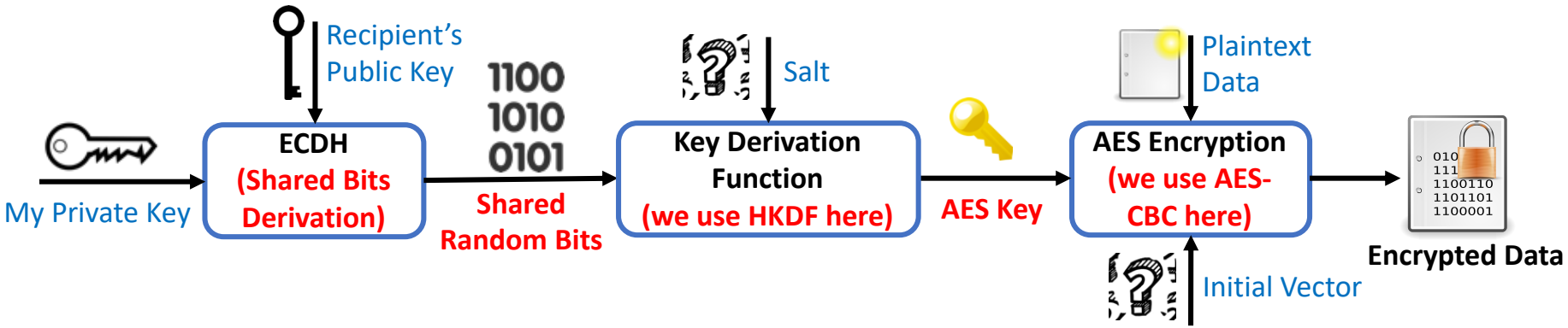
- HKDF (RFC5869)
- Concat KDF (RFC8039)[23]
- etc....

今回は HKDF を使う.

[23] JOSE 向けに標準化された鍵導出関数 https://tools.ietf.org/html/rfc8037

JavaScript で ECDH から AES 暗号化してみよう

ECDH, HDKF, AES を組み合わせて，「公開鍵暗号化」してみる．



このフローでは，実際のデータの暗号化は AES に任せている．

⇓

RSA 暗号化と違い，一度に暗号化できるデータサイズが公開鍵のサイズ以下などと制限されることはない．[24]

[24] ただし，通常この場合も共有ビット列を使って暗号化するデータは，「別の AES 暗号化用の鍵 (すなわち 256bits 以下)」とすることが多い．後述する．

実行してみる.

EC 鍵生成 (2 回やって 2 ペア作る)

```
$ yarn execute ecc-keygen
<Generated ECC Key Pair (DER Form)>
Public Key:
3059301306072a8648ce3d020.....
Private Key:
308193020100301306072a864.....

$ yarn execute ecc-keygen
<Generated ECC Key Pair (DER Form)>
Public Key:
3059301306072a8648ce3d020.....
Private Key:
308193020100301306072a864.....
```

ECDH+HKDF+AES-CBC 暗号化

```
$ yarn execute ecdh-aes-encrypt 'hello world'\
-p '3059301306072a8648ce3d020106082a8648c......' // 送り先の公開鍵
-s '308193020100301306072a8648ce3d0201060......' // 送り元の秘密鍵

<Shared Bits> // ECDH の共有ビット列
51a1a502d01917e6ae0c7cd69cc7078d4a07d0172d271555d001485621551eef

<Derived AES Key> // HDKF で導出した鍵とパラメタ
Key: f52372329867b83ee4e2cada7452a909e85b1ffc2401c5e3b7e7aa7bf9363f7b
HKDF-Salt: 1dffbb6a9a0b91929b690116e3abd75b4a984e4d8686fcd9e35b4bd0220ebfe7
HKDF-Hash: SHA-256

<Encrypted data> // AES-CBC 暗号化したデータ
Data: e36ded44e0e27a8f01d160feb54b1c30
Initial Vector: e9799bff5bdbd3400b1c753e5d5506ff

// Key, Salt, IV, Encrypted Data を Msgpack したやつ
<Msgpacked encrypted and kdf data>
82a9656e6372797074656482a464617461d......
```

パラメタがたくさんでコピペが無理なので Msgpack で serialize して暗号化データを固めている.

ECDH+HKDF+AES-CBC 復号

```
// msgpack したデータを秘密鍵にする.
$ yarn execute ecdh-aes-decrypt '82a9656e63727970746564 82a46461......'
 -p '3059301306072a8648ce3d0201060......' // 送り元の公開鍵
 -s '308193020100301306072a8648ce3......' // 送り先の秘密鍵

<Shared Bits> // ECDH の共有ビット列
51a1a502d01917e6ae0c7cd69cc7078d4a07d0172d271555d001485621551eef

<Derived AES Key> // 共有ビット列と msgpack の中のパラメタから導出した鍵
f52372329867b83ee4e2cada7452a909e85b1ffc2401c5e3b7e7aa7bf9363f7b

<Decrypted Data> // 復号データ
hello world
```

ECDH, HKDF, AES を組み合わせた暗号化のコードはこんな感じ.

暗号化: src/commands-node.js

```
// Shared bits
const sharedBits = await ecdh(publicKeyA, privateKeyB);

// HKDF key derivation
const aesKey = await deriveKeyFromMasterSecret(sharedBits, 32);

// AES-CBC encryption
const encrypted = await encryptAES(data, aesKey.key);

// packing for ease
const packed = msgpack.encode({encrypted, kdfParams: aesKey.kdfParams});
```

復号: src/commands-node.js

```
const depack = msgpack.decode(uint8ArrayData);

// Shared bits
const sharedBits = await ecdh(publicKeyB, privateKeyA);

// HKDF key derivation
const aesKey = await deriveKeyFromMasterSecret(
  sharedBits, 32, depack.kdfParams.salt, depack.kdfParams.hash
);

// AES-CBC decryption
const decrypted = await decryptAES(
  depack.encrypted.data, aesKey.key, depack.encrypted.iv
);
```

楕円曲線の「曲線そのもの」の標準

ここまで，楕円曲線を用いた「ECDH」というデータ処理の手続きの標準について話をしてきた．

だが，この「楕円曲線」そのものにも標準として採択されているものがある．ECDH を運用する際には，この曲線の選択がセキュリティと性能の両側面から重要である[25]．RSA 暗号の「公開鍵ビット長」が「曲線」に相当するイメージ．

※サンプルコードでは P-256 というものをデフォルト値として利用

[25]次週以降話をする楕円曲線 DSA でも同様

RSA 暗号と楕円曲線暗号のパラメタ

- RSA: パラメタは「公開鍵ビット長」
- 楕円曲線: パラメタは「曲線の種類」

曲線の種類は，公開鍵ビット長を定めれば一意に決まるわけではない[26]．

※ただし，RSA 暗号・楕円曲線暗号共々，“一般的には” 公開鍵ビット長が大きいほど，より安全になり，より処理が重くなるというトレードオフがある．

[26] e.g., 公開鍵長 256bit では，P-256 や P-256K など複数の利用可能な曲線の種類が存在．

では，どういった曲線を選べば良いのか？

代表的な楕円曲線パラメタの標準

- SEC2: 業界団体 SECG の標準 [27]
- ANSI X9.62: 米国標準 [28]
- NIST FIPS186-4: 米国標準 [29]

それぞれで標準化されたパラメタは，大きくオーバーラップしている[30]が，最も頻繁に更新されている標準パラメタリストは NIST FIPS 186 のものである．

[27] http://www.secg.org/sec2-v2.pdf

[28] American National Standards Institute, “Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA),” ANSI X9.62, November 2005.

[29] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[30] https://tools.ietf.org/html/rfc8422#appendix-A

WebCrypto API/Node.js 両方で選択できる NIST 曲線パラメタ

- P-256: 公開鍵長 = 256bits, 安全性 ≃ 128bit 鍵 AES
- P-384: 公開鍵長 = 384bits, 安全性 ≃ 192bit 鍵 AES
- P-521: 公開鍵長 = 521bits, 安全性 ≃ 256bit 鍵 AES

Bitcoin Blockchain で使用されている曲線[31]は，WebCrypto API でネイティブには未実装．

[31] SECG パラメタ secp256k1 あるいは P-256K と呼ばれる．

曲線パラメタに対して，より学術的な脆弱性の有無の一覧：
Safe Curves: `https://safecurves.cr.yp.to/`

⇓

現実的な攻撃かどうかはさておき，NIST 曲線パラメタには有効な攻撃が知られている[32]．
実装に不備がある場合に発生する脆弱性が多く，“正しく設計・実装されている既存ライブラリ” の API 経由で利用する分には問題ないと言ってもよいだろう．

[32] safecurves のサイトで `Safe?=False` の場合，攻撃が存在．

より実用的な公開鍵暗号の運用

ここまでの振り返り

ここ迄，以下の 2 つの「単純な」公開鍵暗号のやり方を学んだ．

- RSA-OAEP による暗号化
- ECDH, KDF, AES の組み合わせによる暗号化

ただ，この単純なやり方だと，次のような制限がある．

- 複数宛先向けの暗号化でめちゃくちゃ効率が悪い．
  ※あるデータを，n 人の宛先向けに暗号化すると，トータルの暗号化データは元のデータの n 倍 (以上)．
- 秘密鍵 SK が漏洩すると，何もかも終わり．

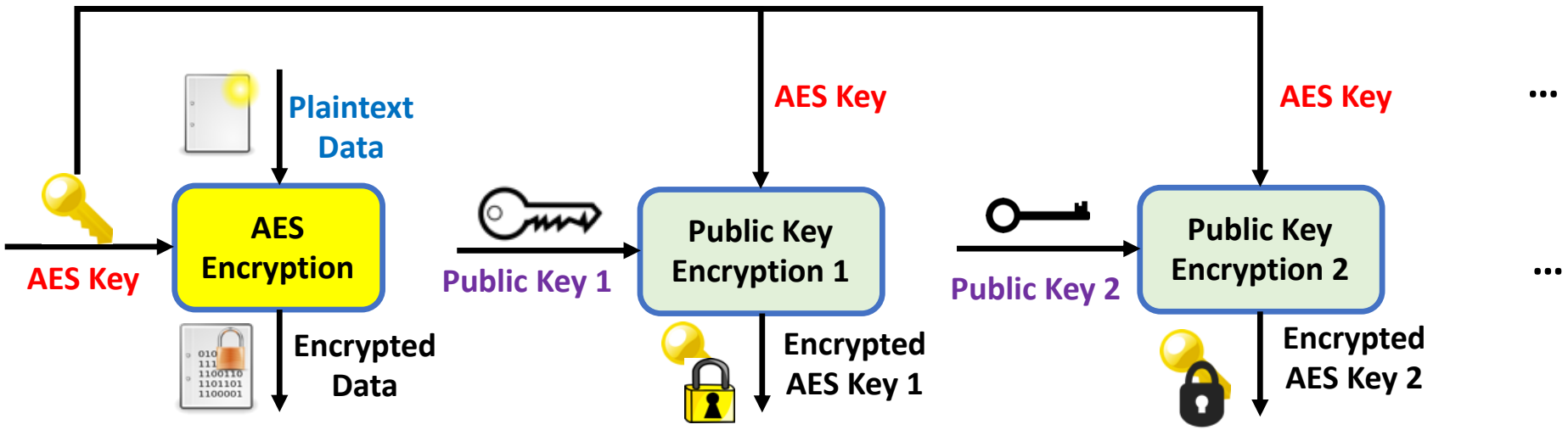
これを踏まえて，ここでそれらの実用的な対策を学ぶ．

1. 公開鍵暗号による AES 暗号鍵のカプセル化
   ⇒ 複数宛先向けに効率のいい公開鍵暗号化．
2. Perfect Forward Secrecy
   ⇒ 秘密鍵が漏洩しても，過去の暗号化データはクラック不能．

公開鍵暗号による，AES 暗号鍵のカプセル化

Hybrid Encryption, Key Encapsulation

- データの暗号化はランダム鍵 K を使った AES 暗号化 [33]
- K のみを宛先ごとに公開鍵暗号化



[33] 共通鍵暗号化

- 比較的小さい AES の鍵 K のみ，宛先毎の公開鍵で暗号化．
- 鍵に比べて非常に大きい (であろう) データ部分の暗号化は，宛先に依らず共通化してしまう．

誰宛でもまず同じ AES 暗号化データを送りつけといて，その AES の鍵だけを復号を許可したい人の公開鍵で暗号化するようなデータへのアクセス制御への応用．

AES 暗号化データ自体は誰宛でも共通なので，CDN に載せて公開配信とかしちゃうとめちゃくちゃ再利用性が高まってよい．[34]

[34] インターネット配信向けの DRM なんかでもこういう考え方がよく使われている．

Perfect Forward Secrecy

より安全な暗号化のために．

(Perfect) Forward Secrecy

長期的に保存されているマスター秘密鍵の漏洩や，一部の暗号化データがクラックされたとしても，それ以外の過去に暗号化されたデータは復号されてしまうことはないという概念．

⇓

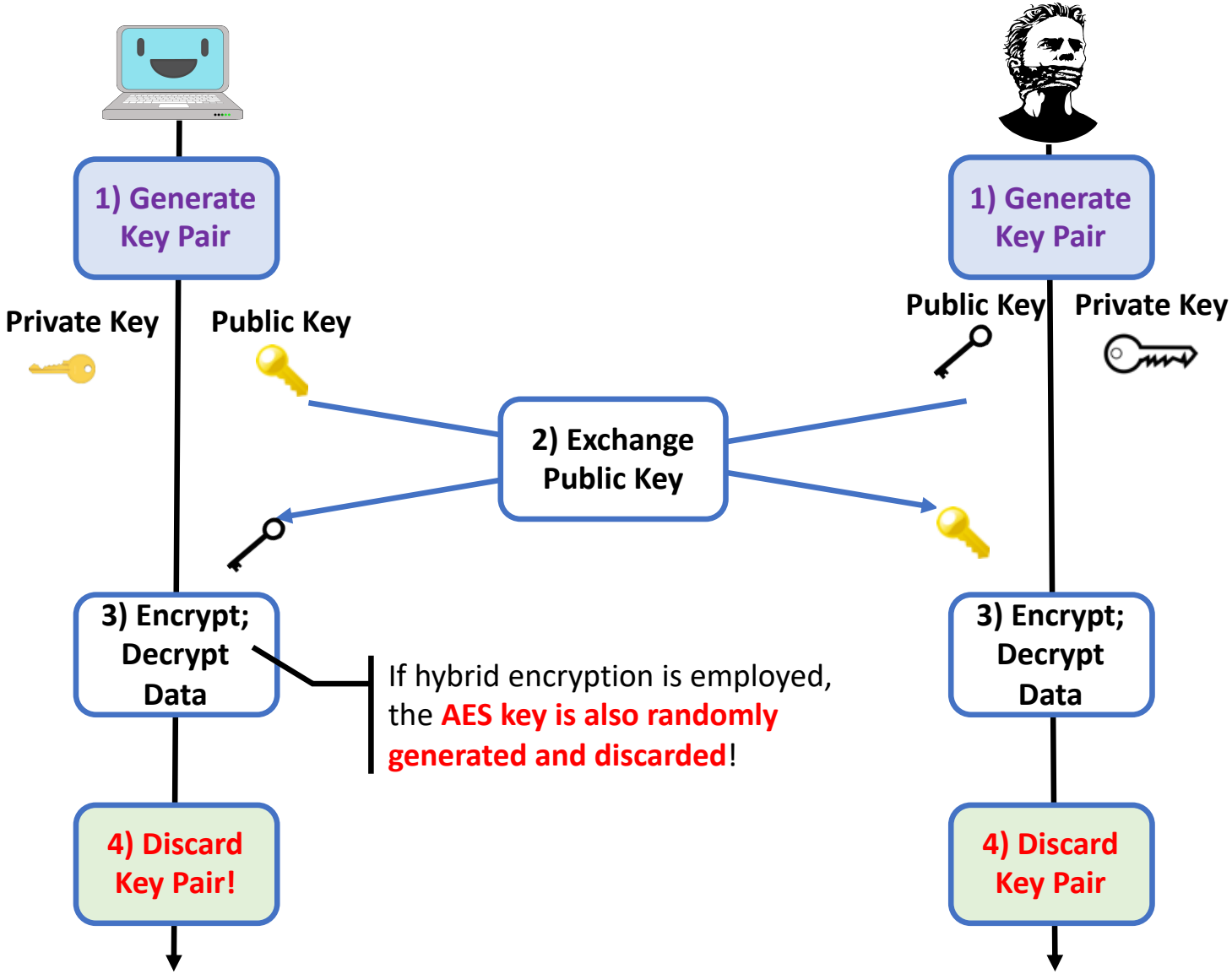
公開鍵・秘密鍵を 1 回限りで使い捨てる Ephemeral Scheme の利用

なんでこんなめんどくさい概念が？

「米国政府機関，SSL/TLS とかの暗号化通信データをそのままガンガン保存してる．保存しておいて，いつか復号できるような機会を待ってる.」とか暴露したことで注目を浴びている．

⇒ 保存データの一箇所でもクラックできたら，一気に他のデータもクラックできるのはよくない．被害は最小限にしよう．

Ephemeral Scheme を用いた暗号化の運用イメージ



ECDHE+AES の場合，共有ビット列から KDF で共有鍵を導出，使い捨て

ECDH を使った暗号化では，Ephemeral Scheme を使うのが推奨される[35]．ECDH Ephemeral は，TLS での鍵交換法 (データそのものの暗号化は別で規定) として RFC4492 → RFC8422 [36]で標準化．

ECDH-Ephemeral (ECDHE)

- 送信先の相手を問わず，毎回違う共有ビット列が生成される．
  ⇒ 「今使った」共有ビット列や秘密鍵が盗まれても過去のデータは解読不能．i.e., Forward secrecy．
- 毎度毎度，まず Ephemeral な公開鍵を送らなければならず，通信コストは高い．
- **「送られてきた Ephemeral な公開鍵は，本当に自分がやりとりしたい相手の公開鍵か?」を確認する手間**もかかる．

ECDHE は TLS1.3 では必須となった．

[35] RSA 暗号を使った場合の Ephemeral Scheme の標準の手法は知る限りない…単純に巨大な公開鍵を送るコストが高いことと，RSA-4096 で AES-256 の鍵を暗号化するなど，無駄の多さが理由か？

[36] https://tools.ietf.org/html/rfc8422

「送られてきた Ephemeral な公開鍵は，本当に自分がやりとりしたい相手の公開鍵か?」を確認する手間もかかる．

⇓

ここでようやく

本人確認・改ざん防止を担保をする「署名」を使う．

次回話す予定．

まとめ

まとめ

- 公開鍵暗号化の際の作法を学んだ．
  - RSA: パディングには OAEP を使う (RSAES-OAEP)．
  - ECDH: 導出した共有ビット列は鍵導出関数を通してから AES の鍵として利用する．
- より実用的な公開鍵暗号化の運用について触れた．
  - ハイブリッド暗号化: 複数人向け，量的に効率的な暗号化．
  - Ephemeral Scheme: Perfect Forward Secrecy を担保して万一の鍵漏洩に備える．

今後の予定 (暫定)

1. 導入&とりあえず暗号化コードを触ってみる
2. 共通鍵暗号化によるデータ秘匿を組み上げる
3. 公開鍵暗号化による鍵秘匿を組み上げる
4. ハッシュ・MAC・署名，それぞれの使い所と使い方 ← 次回
5. 全てをまとめて End-to-End セキュリティを担保する ← 次回