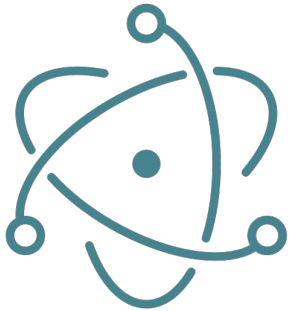
The State of crypto in Node.js

Ujjwal Sharma (@ryzokuken)
ryzokuken@pm.me

“Few false ideas have more firmly gripped the minds of so many intelligent men than the one that, if they just tried, they could invent a cipher that no one could break.”

– David Kahn

About Me



DISCLAIMER: What does “crypto” mean here

```
const crypto = require('crypto');

const tls = require('tls');


crypto === cryptocurrency // => false
```

“Why do I need crypto?”

I am already using TLS!

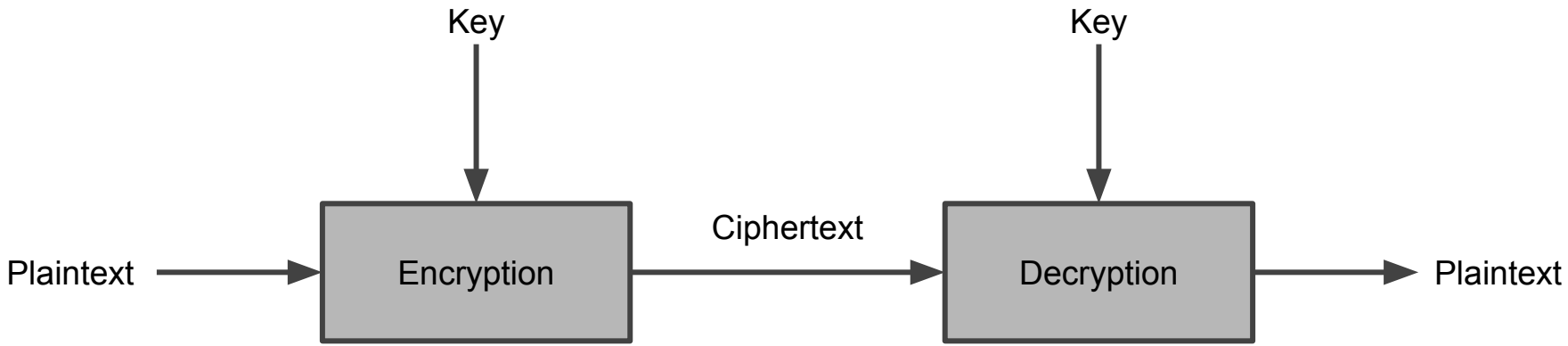
- Encryption
- Key Exchange
- Cryptographic Hashing
- Data Signing
- CSPRNG
- Interoperation

Encryption

- Encoding data to prevent unauthorized access
- “Confidentiality”

- **Cipher** and **Decipher** classes
  - ✘ **createCipher/createDecipher**
  - ✔ **createCipheriv/createDecipheriv**

Encryption: Illustration



Encryption: Example

Encryption

```
const cipher =
crypto.createCipheriv(
    'aes192', K, IV
);

let enc = cipher.update(
    P, 'utf8', 'hex'
);

enc += cipher.final('hex');
```

Decryption

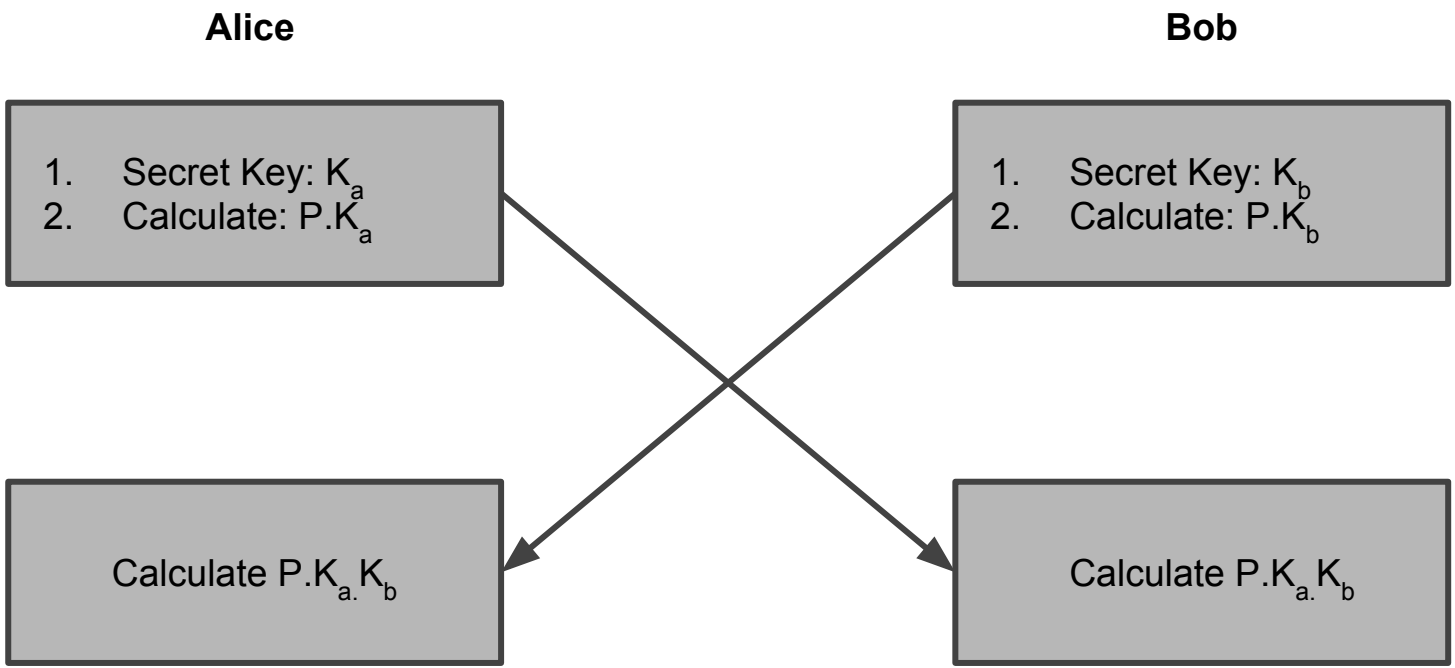
```
const decipher =
crypto.createDecipheriv(
    'aes192', K, IV
);

let dec = decipher.update(
    C, 'hex', 'utf8'
);

dec += decipher.final('utf8');
```

Key Exchange

- Securely exchanging keys over a public channel
- One way to agree upon a key before a conversation

- Two types
  a. Prime Number-based
  b. Elliptic Curve-based
- **DiffieHellman** and **ECDH** classes respectively

Key Exchange: Illustration



Key Exchange: Example (DiffieHellman)

———

```
const alice = crypto.createDiffieHellman(2048);
const bob = crypto.createDiffieHellman(
    alice.getPrime(), alice.getGenerator()
);

const aliceKey = alice.generateKeys();
const bobKey = bob.generateKeys();

const aliceSecret = alice.computeSecret(bobKey);
const bobSecret = bob.computeSecret(aliceKey);

assert.strictEqual(aliceSecret.toString('hex'), bobSecret.toString('hex')); ✔
```

Key Exchange: Example (ECDH)

———

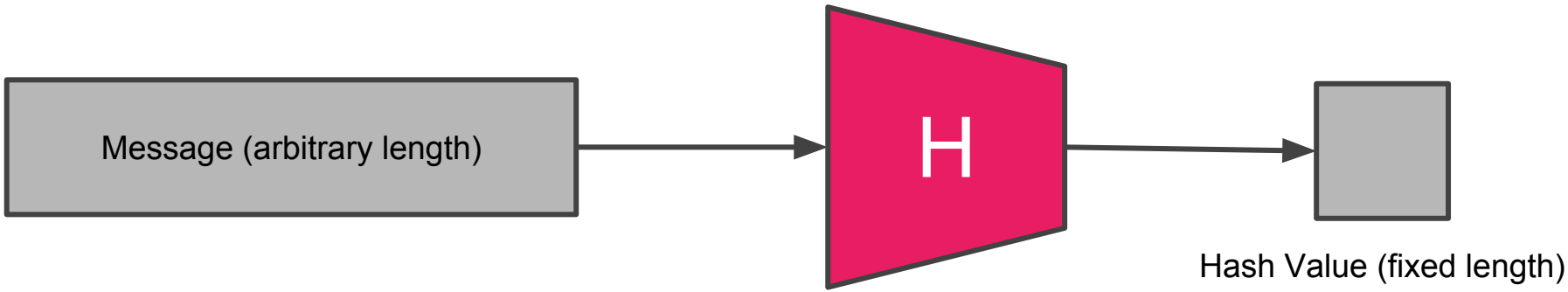
```
const alice = crypto.createECDH('P-256');
const bob = crypto.createECDH('P-256');

const aliceKey = alice.generateKeys();
const bobKey = bob.generateKeys();

const aliceSecret = alice.computeSecret(bobKey);
const bobSecret = bob.computeSecret(aliceKey);

assert.strictEqual(aliceSecret.toString('hex'), bobSecret.toString('hex')); ✔
```

Hashing

- Hash Functions
  - Map arbitrarily sized data to fixed-sized bit strings (hash)
  - Hard to invert, collision-resistant
- “Authentication” and “Integrity”
- Data Signing, HMACs, etc

- **Hash** class
- **Hmac** class

Hashing: Illustration



Hashing: Example

Hash

```
const hash = crypto.createHash(
    'sha256'
);

hash.update('some data to hash');
console.log(hash.digest('hex'));
```

Hmac

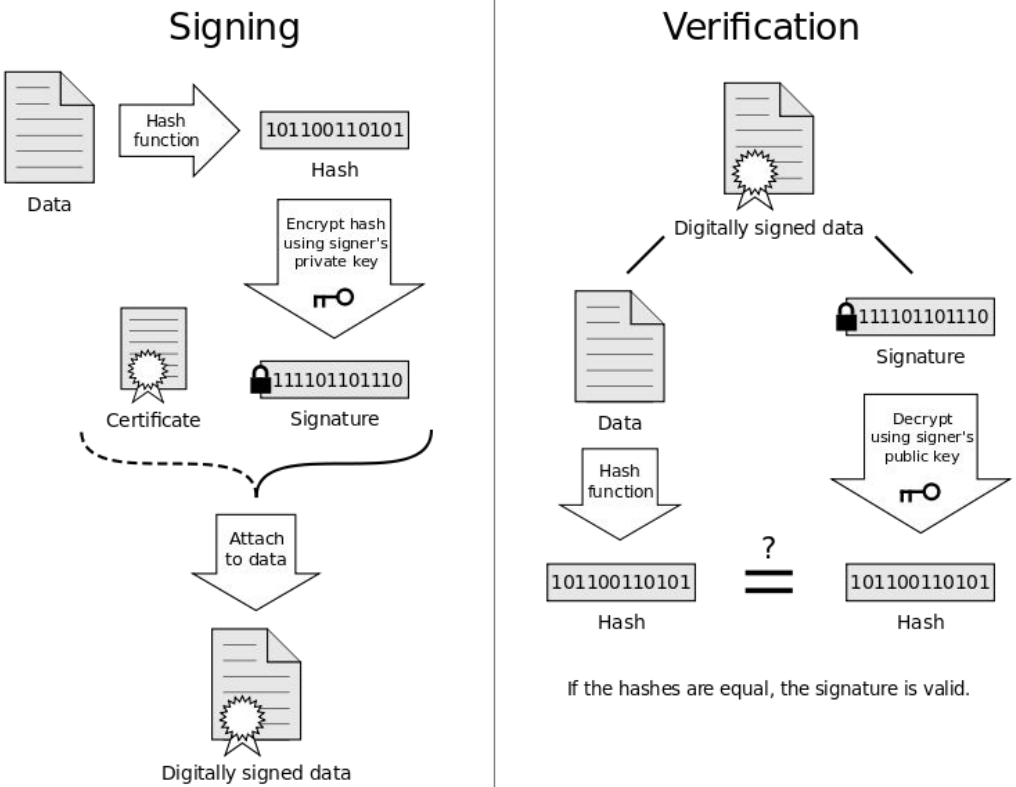
```
const hmac = crypto.createHmac(
    'sha256', ‘a secret’
);

hmac.update('some data to hash');
console.log(hmac.digest('hex'));
```

Data Signing

- Presenting authenticity of digital messages and documents
- “Authentication”, “Non-Repudiation”, “Integrity”

- **Sign** and **Verify** classes

Data Signing: Illustration



Source: Wikimedia Commons

Data Signing: Example

Sign

```
const sign =
crypto.createSign('SHA256');

sign.update('some data to sign');

const privateKey = getPrivateKey();

const signature =
    sign.sign(privateKey, 'hex');
```

Verify

```
const verify =
crypto.createVerify('SHA256');

verify.update('some data to sign');

const publicKey = getPublicKey();

const result = verify.verify(
    publicKey, signature
); // => true
```

Pseudo-random Number Generator

- Cryptographic applications require random numbers
  - Key Generation
  - Initialization Vectors
  - Salts
  - One-time pads, Claude Shannon’s perfect secrecy
- Need for higher entropy

- **randomBytes** and **randomFill** functions

“The generation of random numbers is too important to be left to chance.”

– Robert R. Coveyou

PRNG: Example

---

randomBytes

Synchronous

```
const buf = crypto.randomBytes(256);
console.log(buf);
```

Asynchronous

```
crypto.randomBytes(256, (e, buf) => {
    console.log(buf);
});
```

randomFill

Synchronous

```
const buf = Buffer.alloc(256);
const fb = crypto.randomFill(buf);
console.log(fb.toString('hex'));
```

Asynchronous

```
const buf = Buffer.alloc(256);
crypto.randomFill(buf, (e, buf) => {
    console.log(buf.toString('hex'));
});
```

Interoperability

- Many great crypto libraries
  - Bouncy Castle (Java, C#)
  - NaCL (C)
  - libsodium (C)
  - PyCryptodome (Python)
  - …
- Node.js uses OpenSSL
- WebCrypto
- BoringSSL, Chromium, Electron

Why crypto isn’t easy

A justification

- Simplicity
- Conventions
- Security and Safety
- Compatibility
- Feature-completeness

WebCrypto Compatibility

- WebCrypto - JavaScript API for crypto stuff
- High-value target for interoperability

- Different goals and values – “It’s JavaScript, right?”
- Road to interoperability
  - Key Generation - DER vs PEM (@tniessen)
  - Key Objects
  - ...

Side-channel attacks

- Not based on the weakness of the implemented algorithm
- Based on the information gained from the implementation

- Cache attack
- Timing attack
  - crypto.**timingSafeEqual**(a, b)
- Power-monitoring
- ...

Don’t roll out your own crypto

The beauty and pragmatism behind using OpenSSL

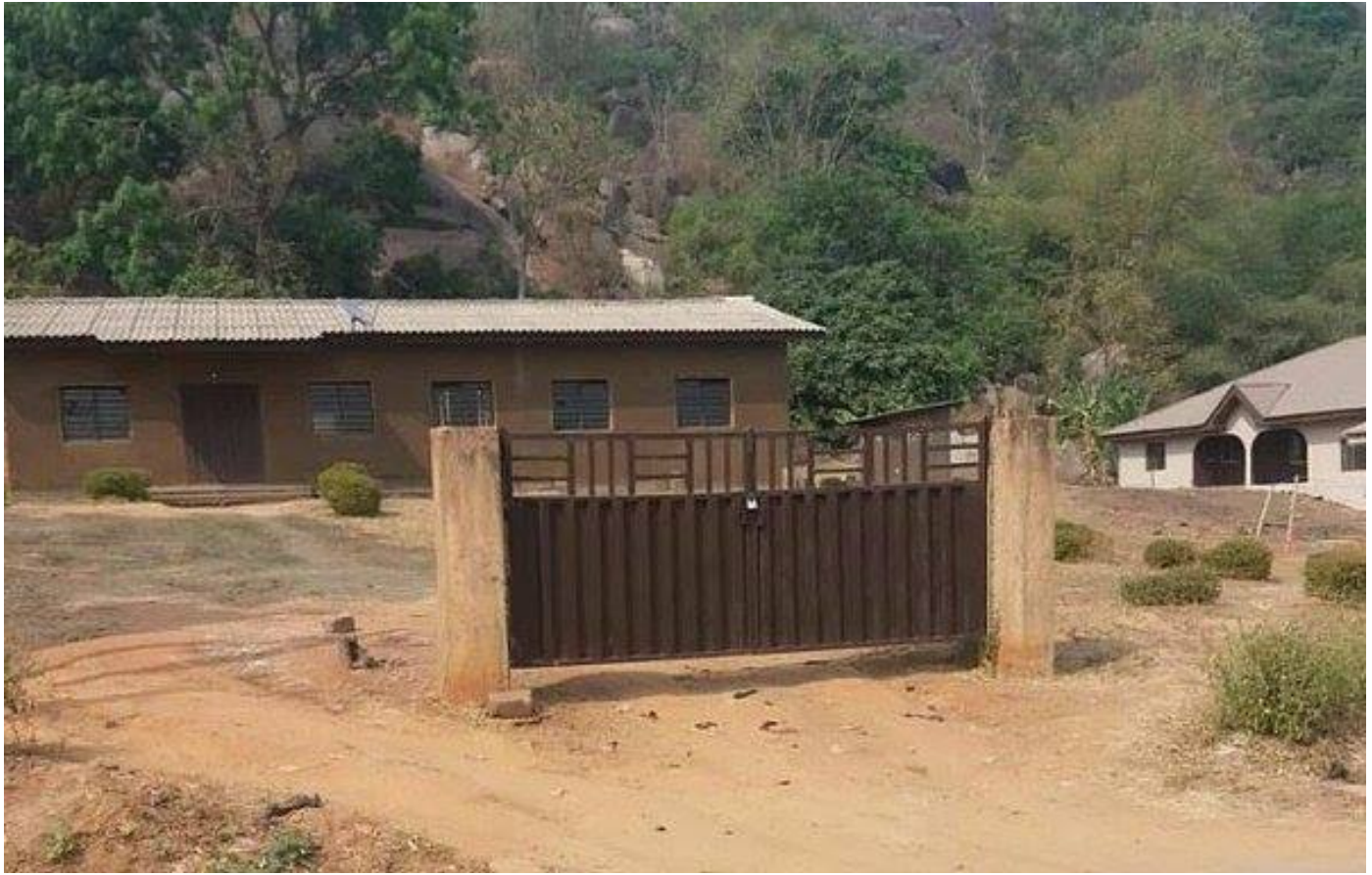
- Building cryptosystems is hard
- Building *secure* cryptosystems is harder
- Building on top of a battle-tested foundation: OpenSSL

“Did I ever tell you about the do-it-yourself brain surgery I performed on my late mother-in-law? Everything went fine until she went and died. ”

– Bob Bryan

Homecooked Crypto Hall of Shame

- IOTA - Curl
- Telegram
  - MTProto
  - Encrypted Profiles
- MIFARE - Crypto1
- SaltStack - RSA
- WEP



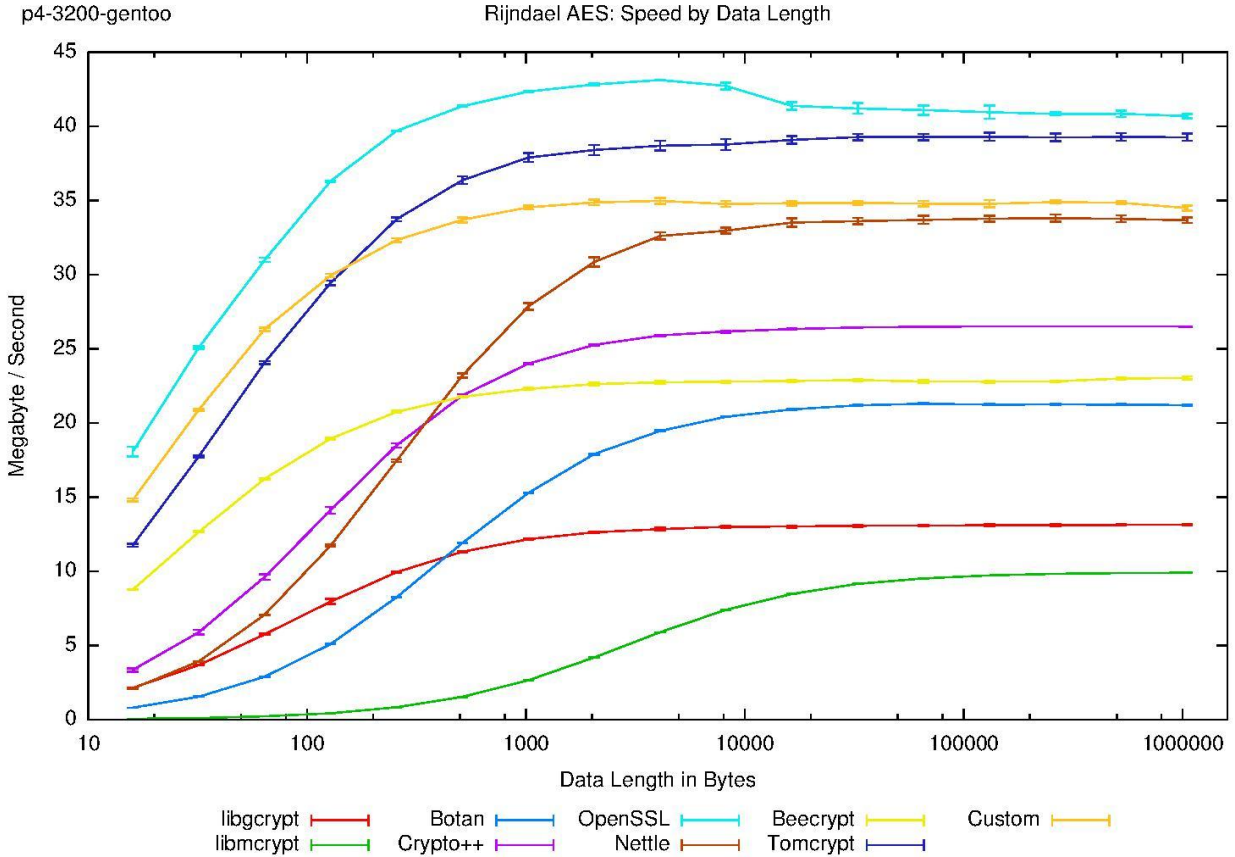
IT'S OVER 9000!

OpenSSL

- OpenSSL is a **robust**, **commercial-grade**, and **full-featured** toolkit for the TLS and SSL protocols. It is also a general-purpose cryptography library.
- Built and scrutinized by experts
- Trust the ecosystem
  - Python's **hashlib**
  - Ruby's **openssl**
- Please don't build your own: Go, Rust

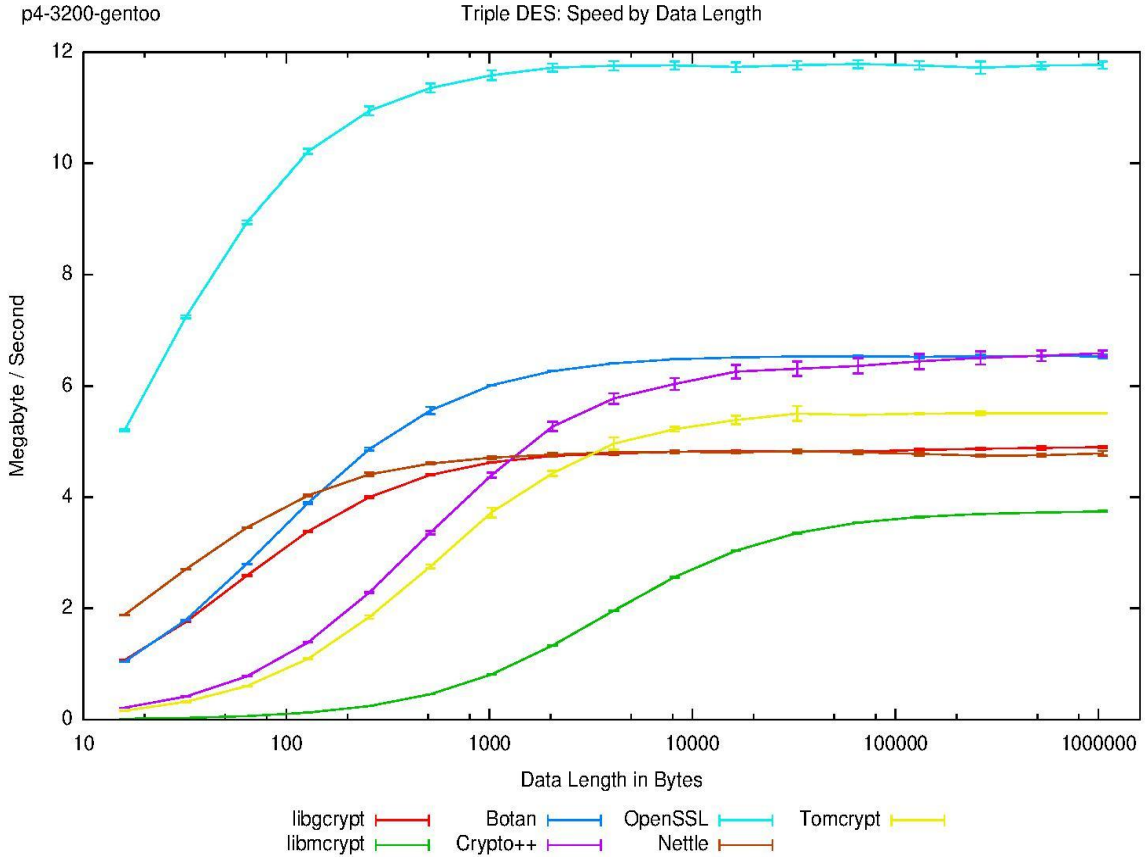
Also, it’s fast.

Like... *Really* fast.



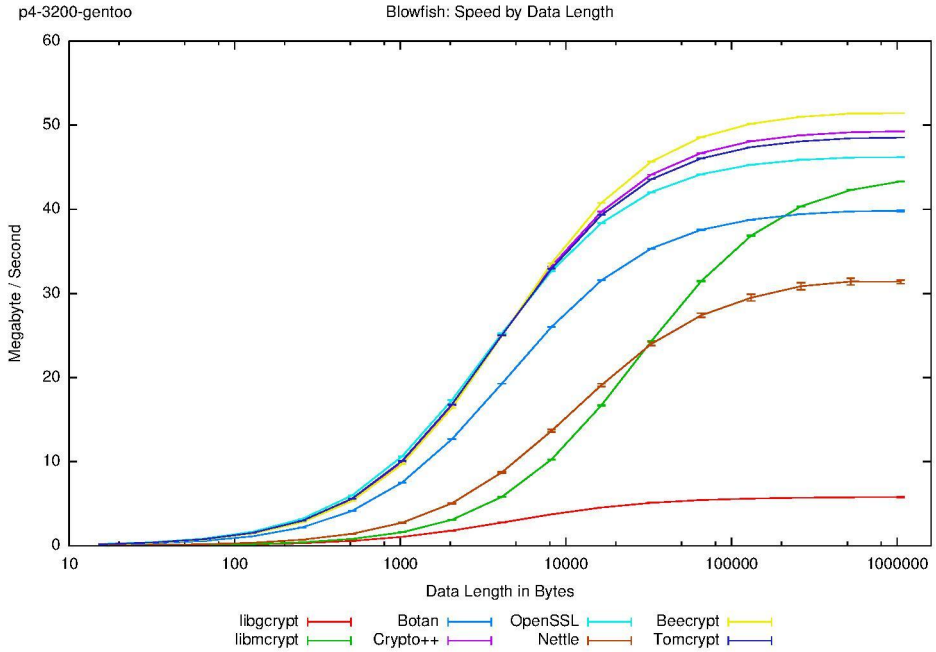
Fastest for AES

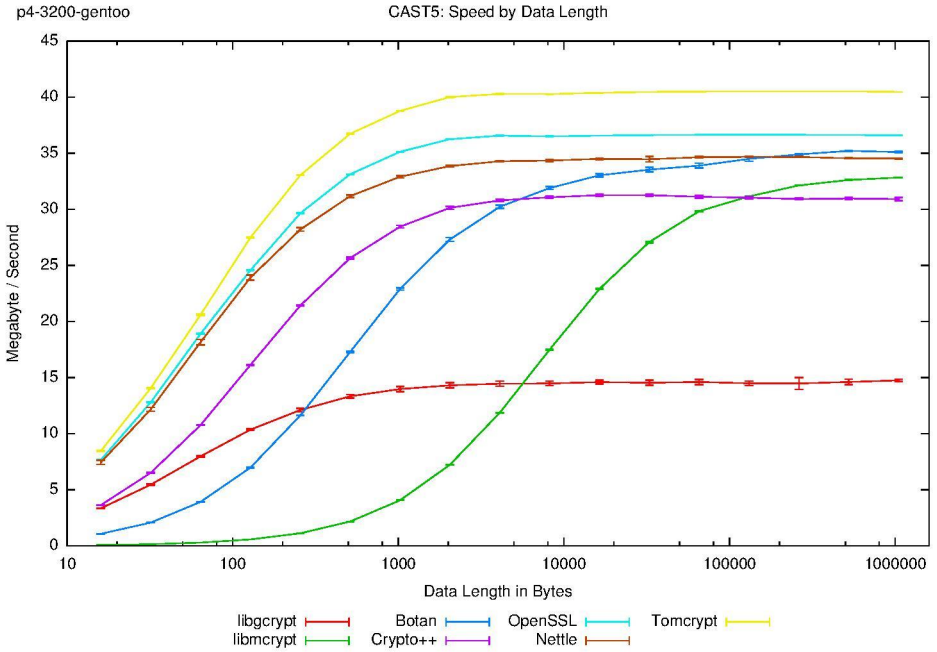
Source: https://panthema.net/2008/0714-cryptography-speedtest-comparison/



Fastest for 3DES. By a fair margin.

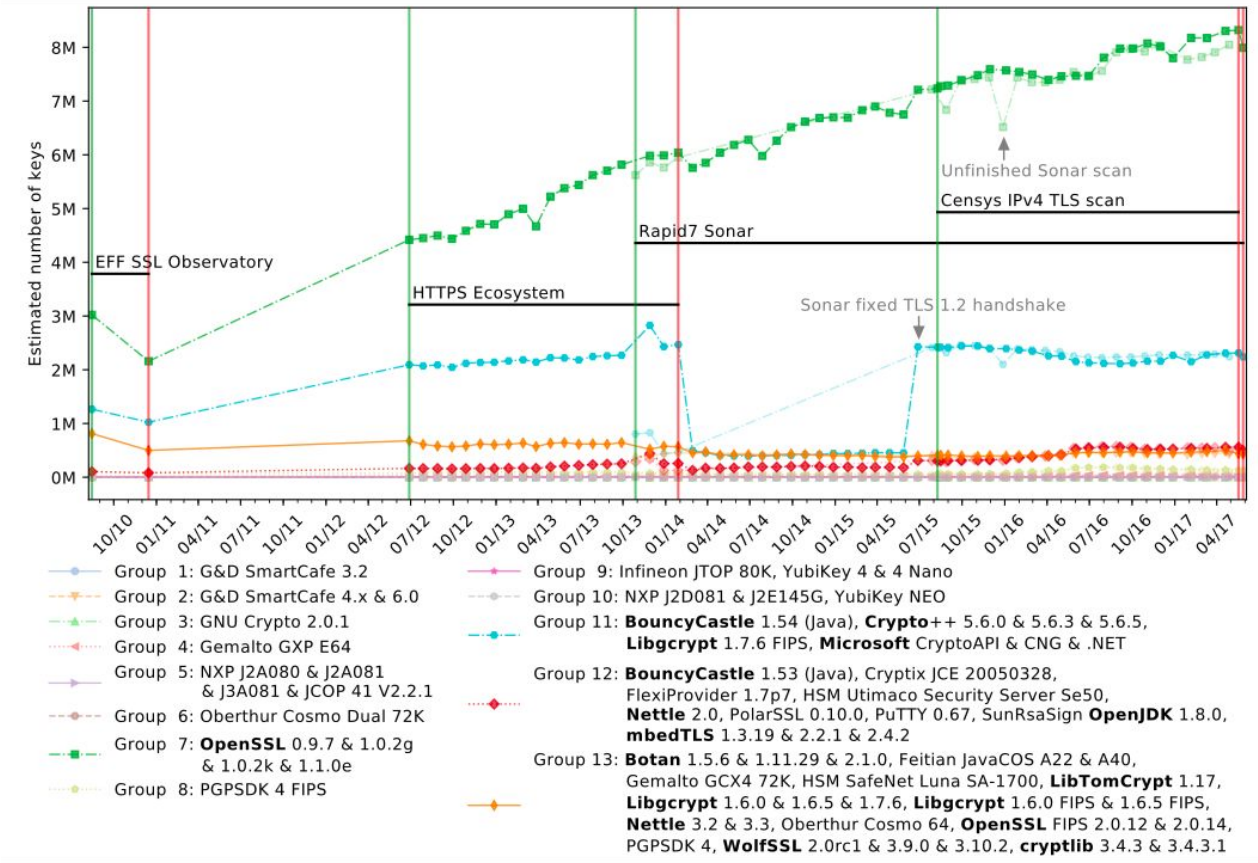
Source: https://panthema.net/2008/0714-cryptography-speedtest-comparison/





Pretty damn fast for everything else.

Source: https://panthema.net/2008/0714-cryptography-speedtest-comparison/



Did I tell you it was also the most popular?

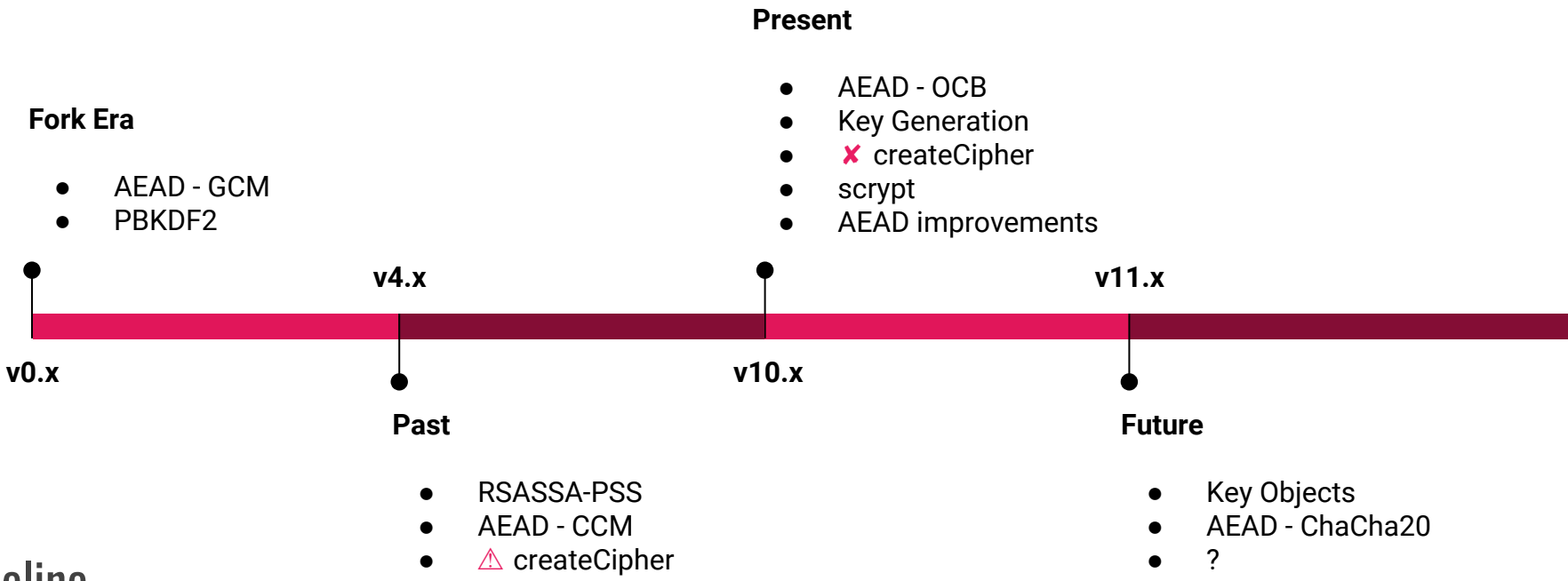
Source: https://crocs.fi.muni.cz/public/papers/acsac2017

The State of crypto

The past, the present and the future of the module

- AEAD: GCM, CCM, OCB
- AEAD: Improvements
- Key Generation
- createCipher **deprecated**
- Password-based KDFs
- RSASSA-PSS

Timeline

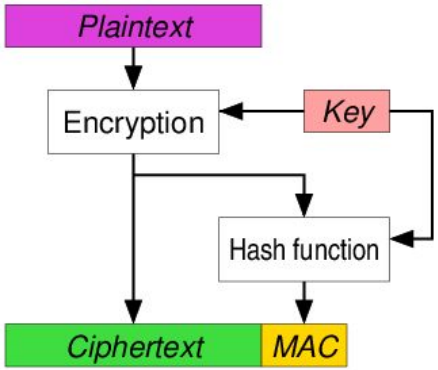

Fork Era
AEAD - GCM
PBKDF2
v0.x
v4.x
Past
RSASSA-PSS
AEAD - CCM
⚠ createCipher
Present
AEAD - OCB
Key Generation
✘ createCipher
scrypt
AEAD improvements
v10.x
v11.x
Future
Key Objects
AEAD - ChaCha20
?

AEAD and AAD

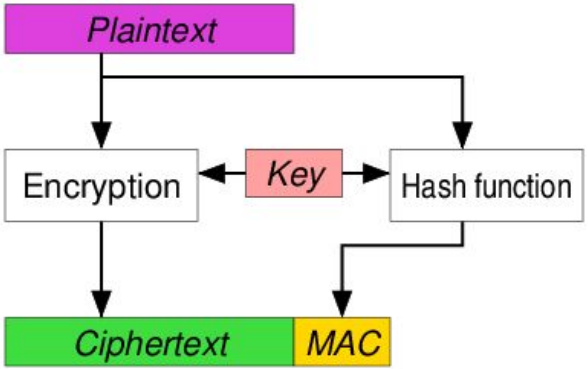
- Encryption + Authentication = AEAD (or AE)
- Way too common to be done separately
- cipher.**getAuthTag**() and decipher.**setAuthTag**()
- cipher.**setAAD**() and decipher.**setAAD**()

- Cipher modes
  - GCM (@KiNgMaR) v0.11.10
  - CCM (@tniessen) v10.0.0 (semver-major)
  - OCB (@tniessen) v11.0.0 (backported to v10.10.0)
  - ChaCha20-Poly1305 (@chux0519) Coming soon to a binary near you

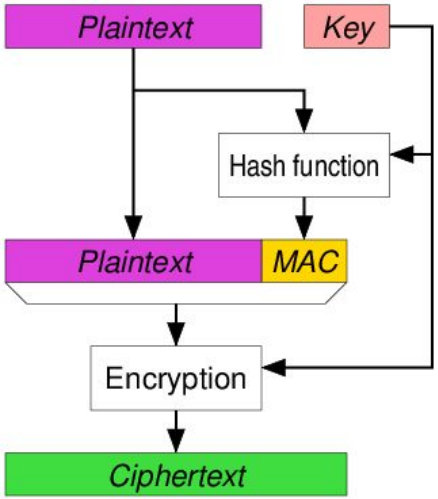
Encrypt-then-MAC



Encrypt-and-MAC



MAC-then-Encrypt



Source: Wikipedia

Improvements to Authenticated Encryption

- Allow to restrict valid GCM tag length
  - Narrow down the list of valid tag lengths to a single value
  - @tniessen, v10.12.0
- Allow to produce GCM tags with a specific length
  - Add support for **authLengthTag** option
  - @tniessen, v10.12.0
- Disallow multiple calls to **setAuthTag**
  - Makes no sense whatsoever
  - Makes it hard to detect bugs
  - @tniessen, v11.0.0

Key Generation

- Asymmetric key generation
- Supports multiple cryptosystems
  - RSA (prime number based)
  - DSA (discrete log based)
  - EC (elliptic curve based)
- Supports DER alongside PEM (WebCrypto compatibility)
- **generateKeyPair** and **generateKeyPairSync** functions
- @tniessen, v11.0.0 (backported to v10.12.0)

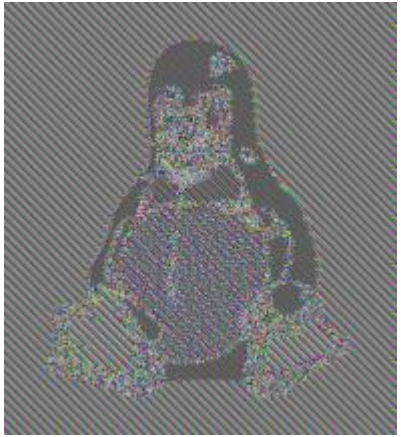
Key Generation: Support Matrix

	RSA 🔊	RSA □	DSA 🔊	DSA □	EC 🔊	EC □
PKCS#1	✔	✔	✘	✘	✘	✘
PKCS#1 🔒	✘	✔	✘	✘	✘	✘
SPKI	✔	✘	✔	✘	✔	✘
PKCS#8	✘	✔	✘	✔	✘	✔
PKCS#8 🔒	✘	✔	✘	✔	✘	✔
SEC1	✘	✘	✘	✘	✘	✔
SEC1 🔒	✘	✘	✘	✘	✘	✔

“IV or not to be, that
is the question.”
– Prince Hamlet

createCipher and createDecipher deprecation

- Added security, randomness
- **createCipher** may be insecure
- Don’t use CTR, GCM or CCM modes
- IV reuse causes vulnerabilities



- Use **createCipheriv** and **createDecipheriv**
- doc-only deprecation in v10.0.0 (@tniessen)
- runtime deprecation in v11.0.0 (@tniessen)

Source: Wikipedia

Password-based Key Derivation Functions

- Deriving a *secure* key using an insecure password
- Makes password cracking difficult — “key stretching”

- Supported functions
  - PBKDF2 (@pixelglow) forever: v0.6.0
  - scrypt (@bnoordhuis) v10.5.0
- Userland: @joepie91’s **scrypt-for-humans**

RSASSA-PSS

- “Probabilistic” padding scheme
- Like normal RSA, but only for signatures
- Stronger than PKCS#1 v1.5
- Security reducible to the RSA problem

- Alternative to PKCS#1 v1.5 in the **Sign** and **Verify** classes
- @tniessen, v8.0.0

Special Thanks

- Tobias Niessen (@tniessen)
- Ben Noordhuis (@bnoordhuis)
- Sven Slootweg (@joepie91)
- Anna Henningsen (@addaleax)
- Shelley Vohr (@codebytere)

- Node.js core collaborators
- HolyJS organizers

THANK YOU