OWASP
BLINDSPOT

Cryptography 101 - Part 2

When Good Crypto Goes Bad

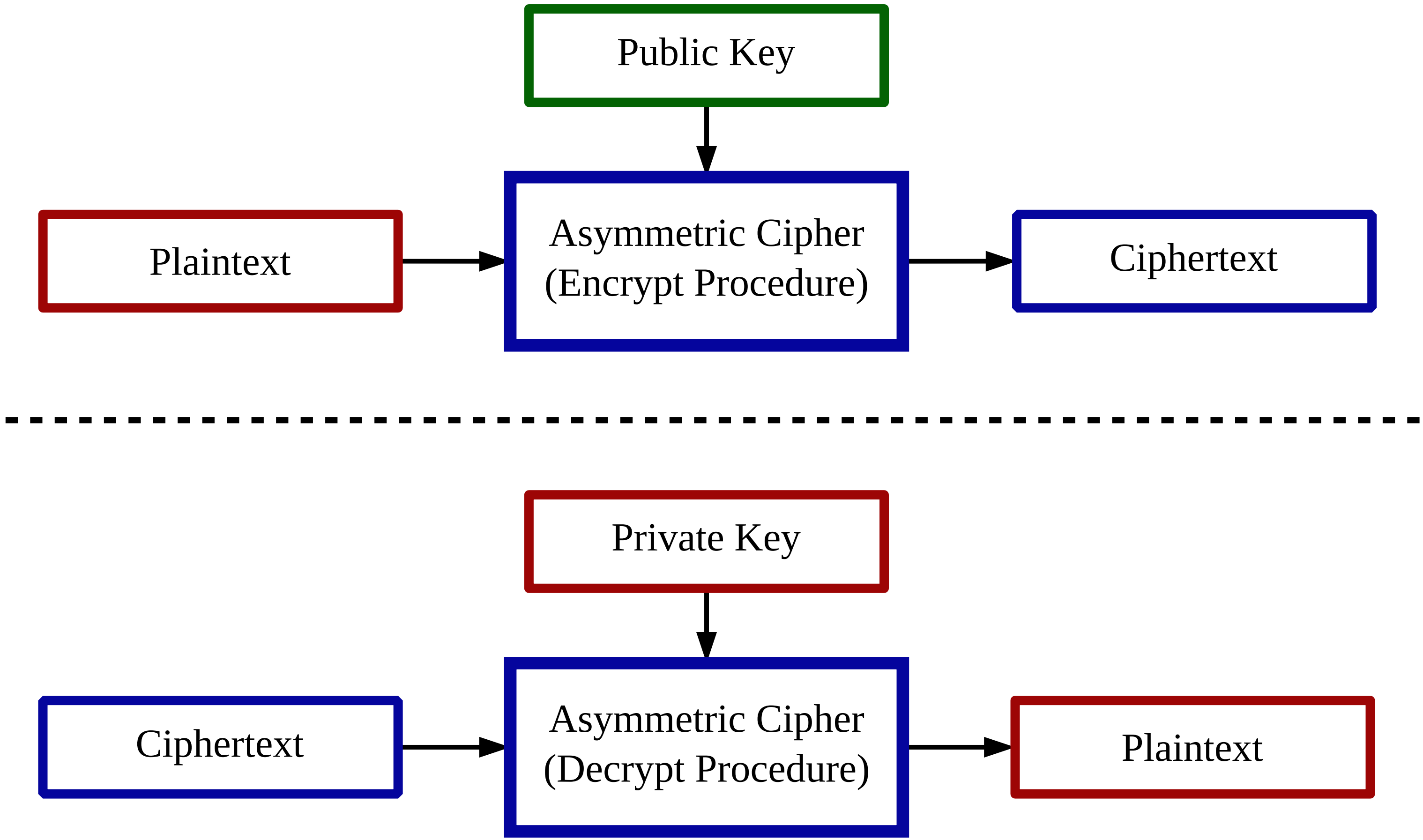
OWASP Chapter Meeting - November 14, 2017

Timothy D. Morgan (@ecbftw)

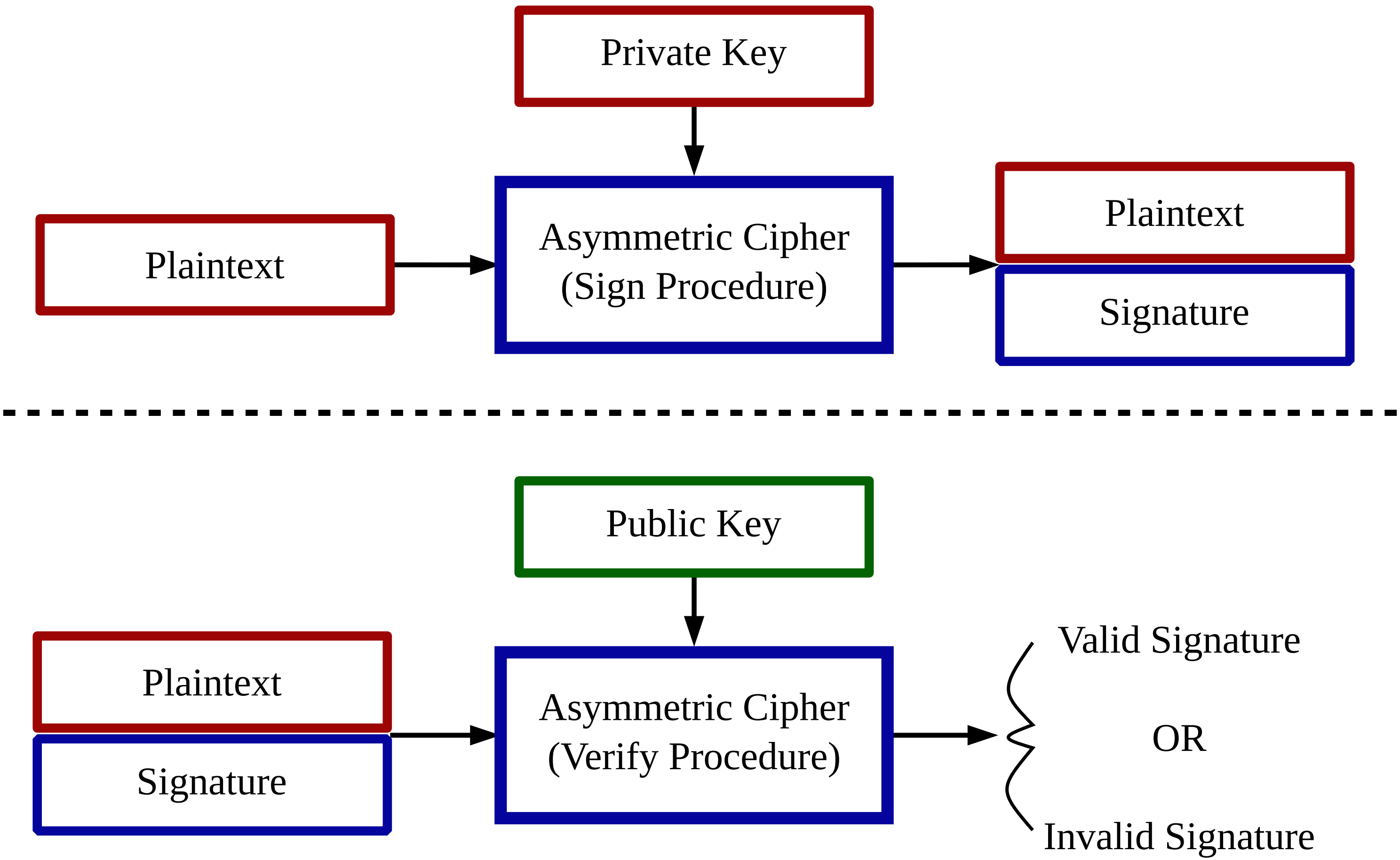
Overview

- Teaser: Breaking AES
- How Public Key Usually Fails
- How Cryptographers Got Symmetric Key Wrong
- Demo: Analyzing Ciphertext

Asymmetric Ciphers Illustrated



Asymmetric Signatures Illustrated

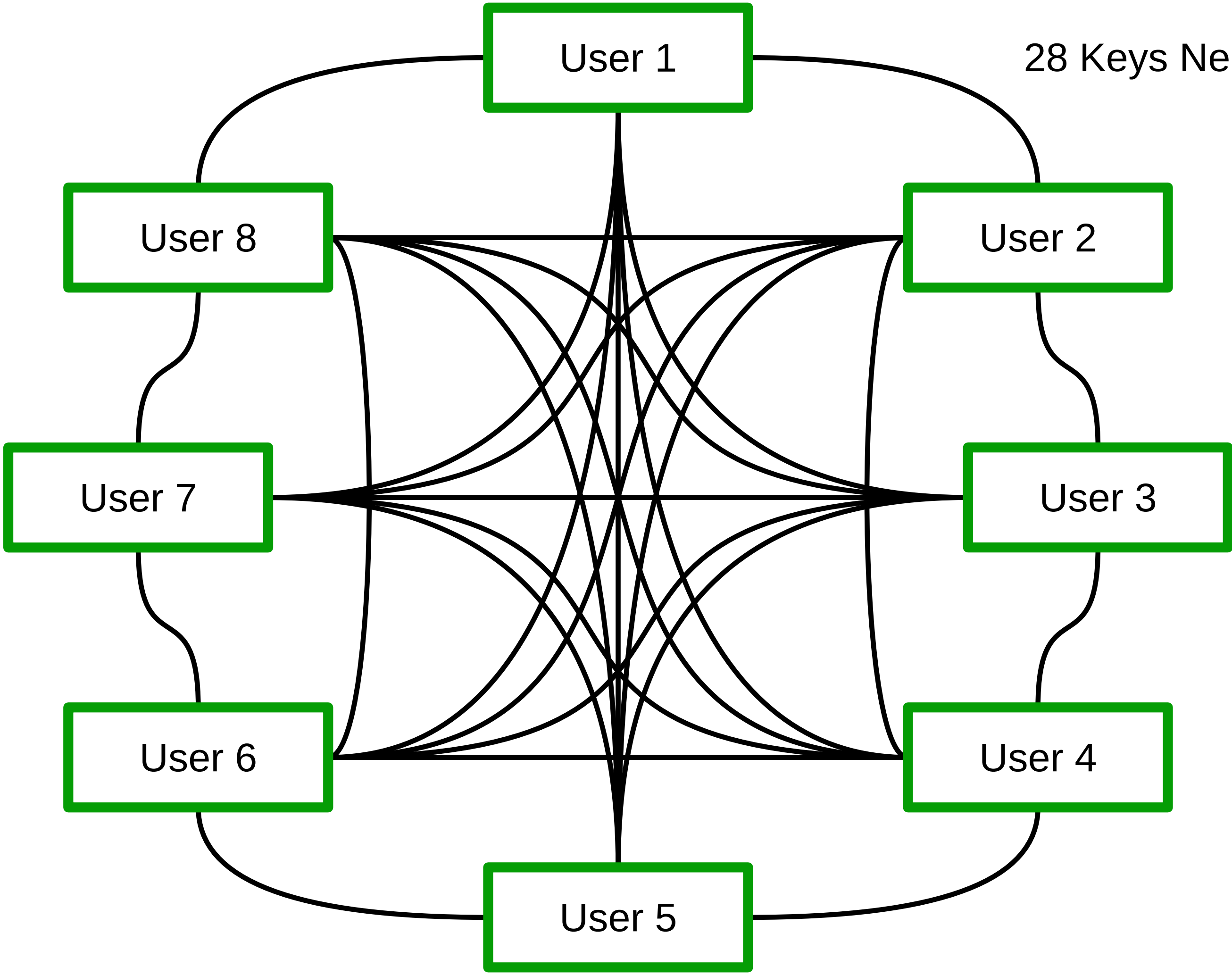


Asymmetric Benefits

- Public key encryption revolutionized the Internet
- Suppose you have a group of `N` people
    - From time to time, pairs of people want to hold private conversations
    - If you used symmetric key encryption, you would need `(N*(N-1)/2)` keys, all communicated *secretly*
    - With asymmetric algorithms, you need just `N` key pairs, communicated with *known identity*

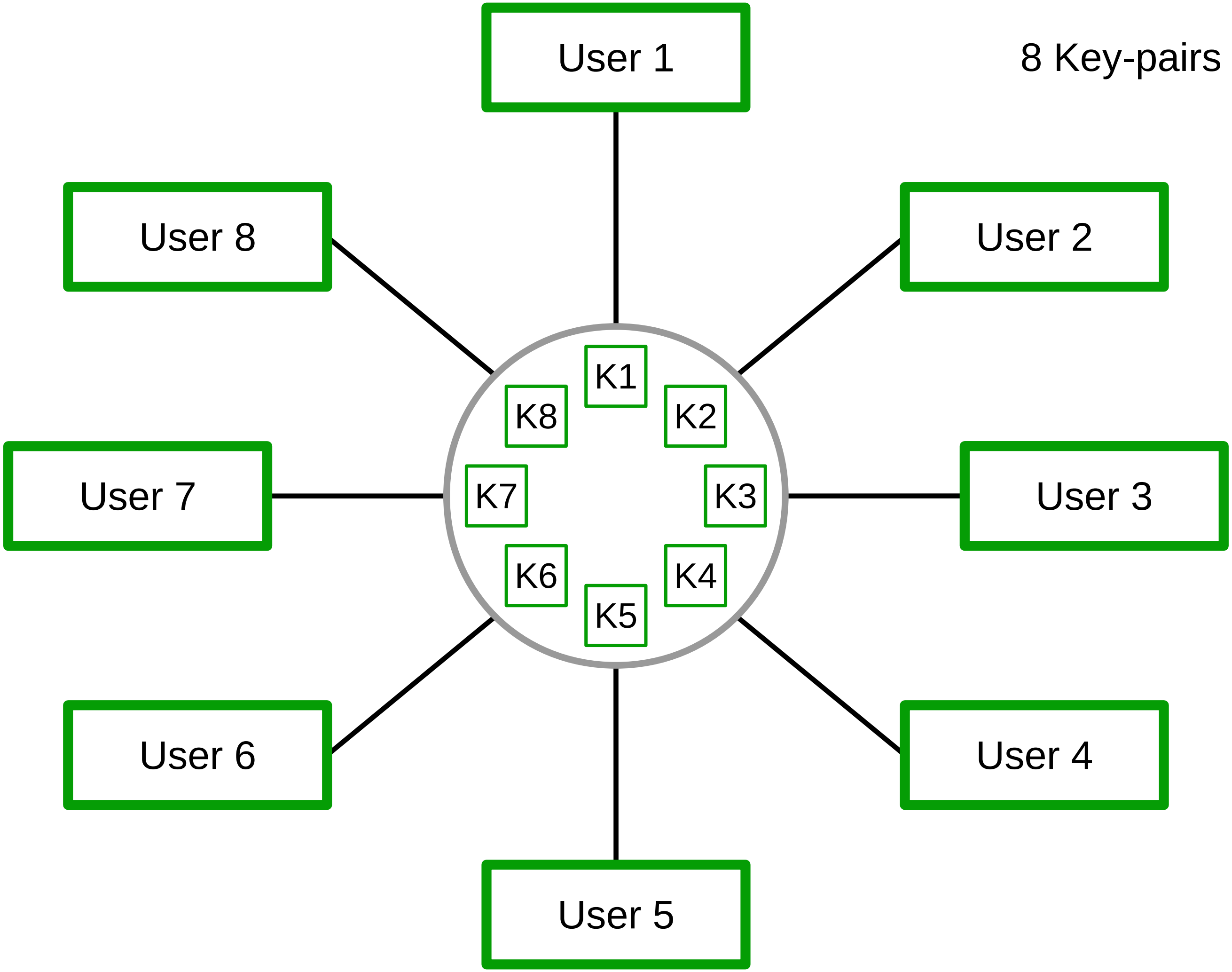
Symmetric Encryption in a Group

28 Keys Needed

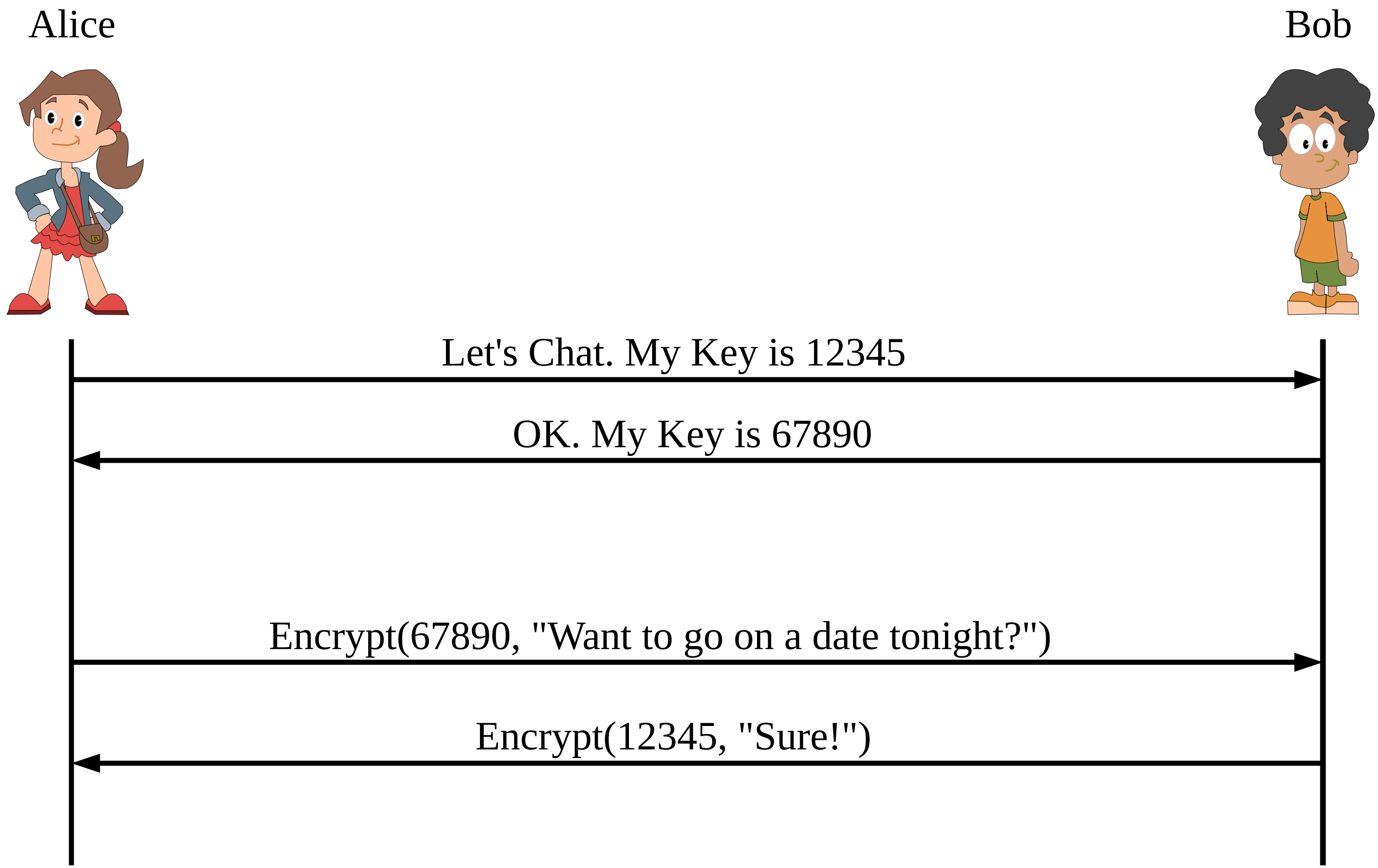


Asymmetric Encryption in a Group

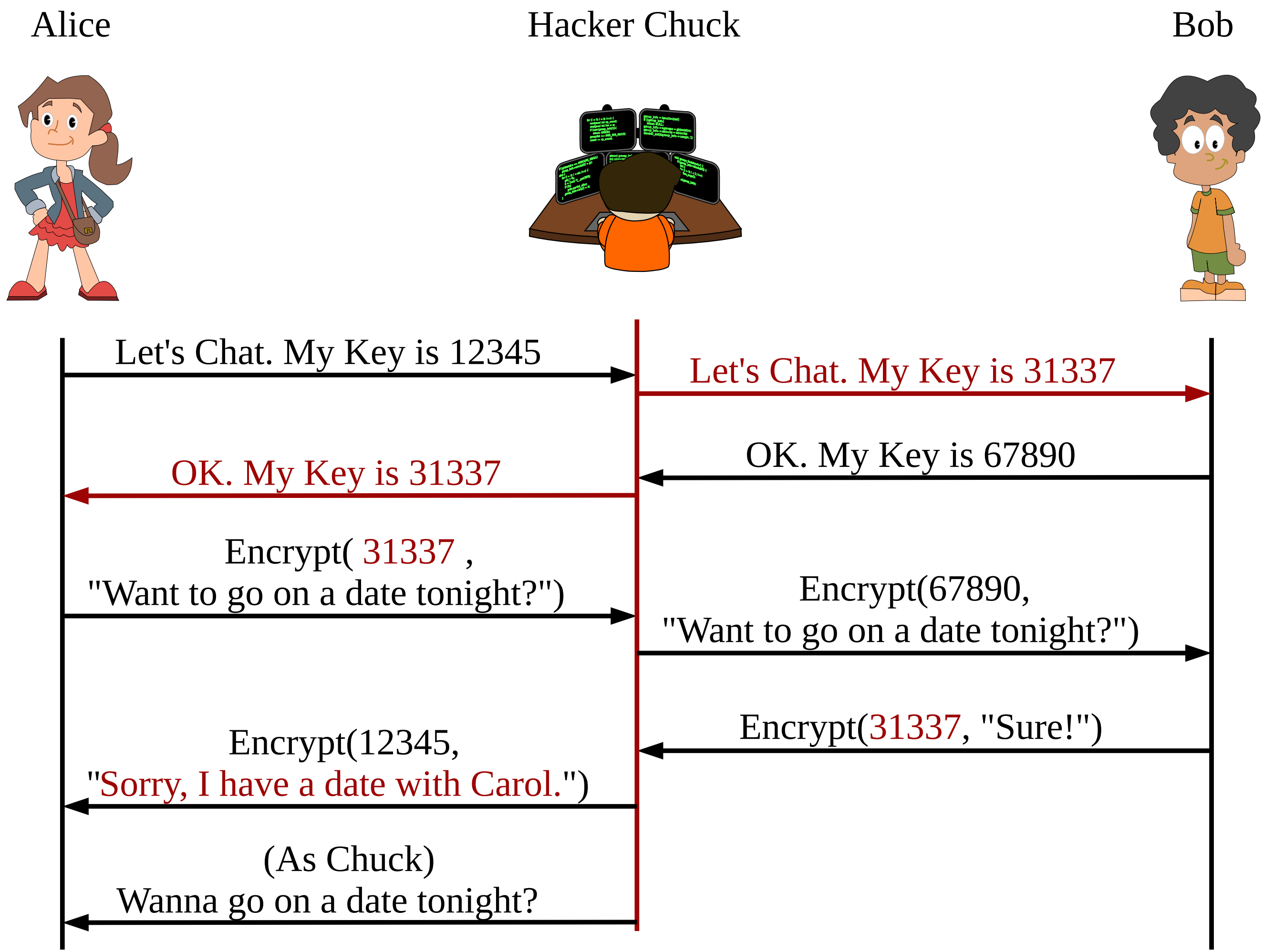
8 Key-pairs Needed



Insecure Key Exchange



Cryptographic MitM



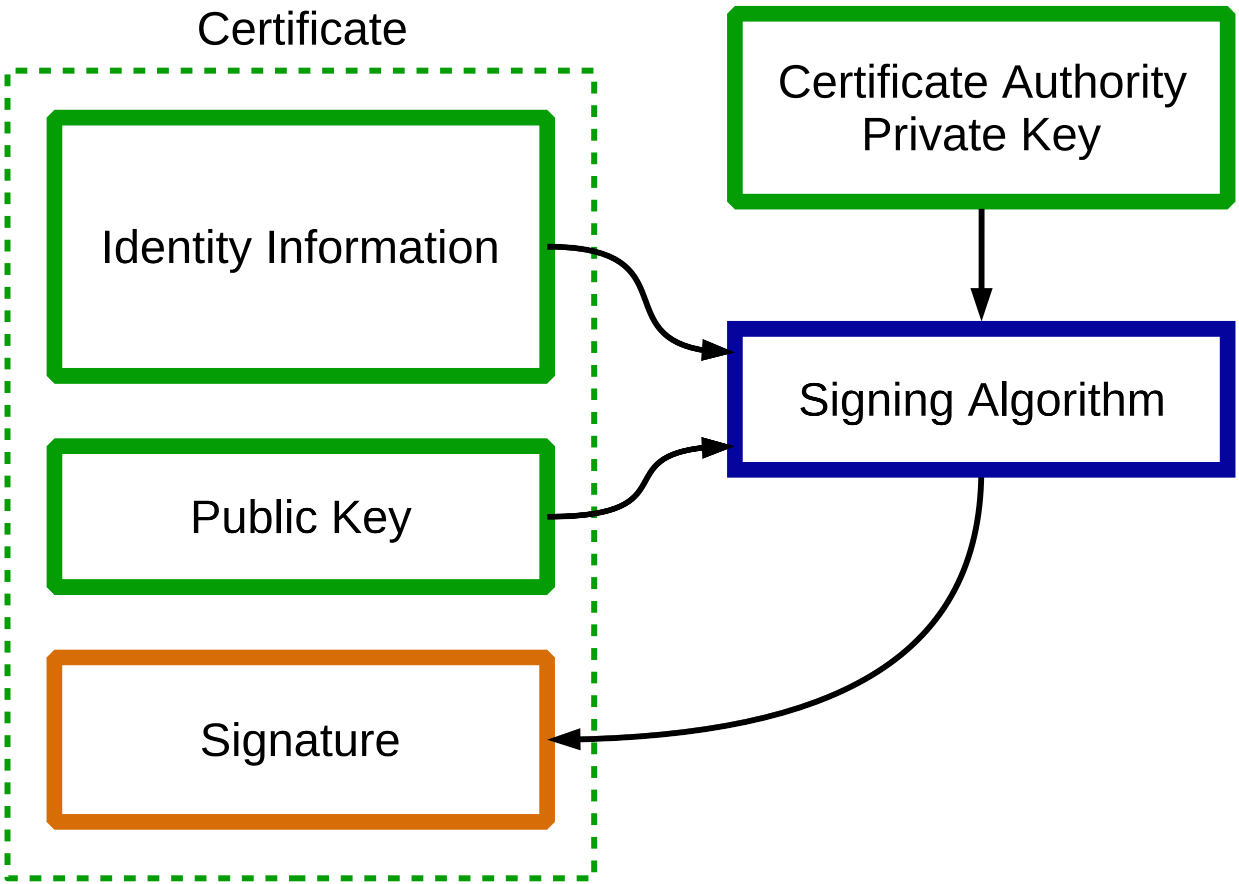
Trusting Keys

- Must somehow tie each public key to an **identity**
- But what is an identity?
  - People care about: name, address, driver's license number, SSN, ...
  - Computers care about: domain names, email addresses, IP addresses, ...
  - Specific definition depends on situation and communications medium
  - Better if it is end-to-end, but may be impractical

Tying Identities to Keys

- Hard-Coding/Certificate Pinning
- Trust-on-first-use
- Certificates and public key infrastructures (PKI)
  - Hierarchical (like SSL/TLS, S/MIME)
  - Web of trust (like PGP)

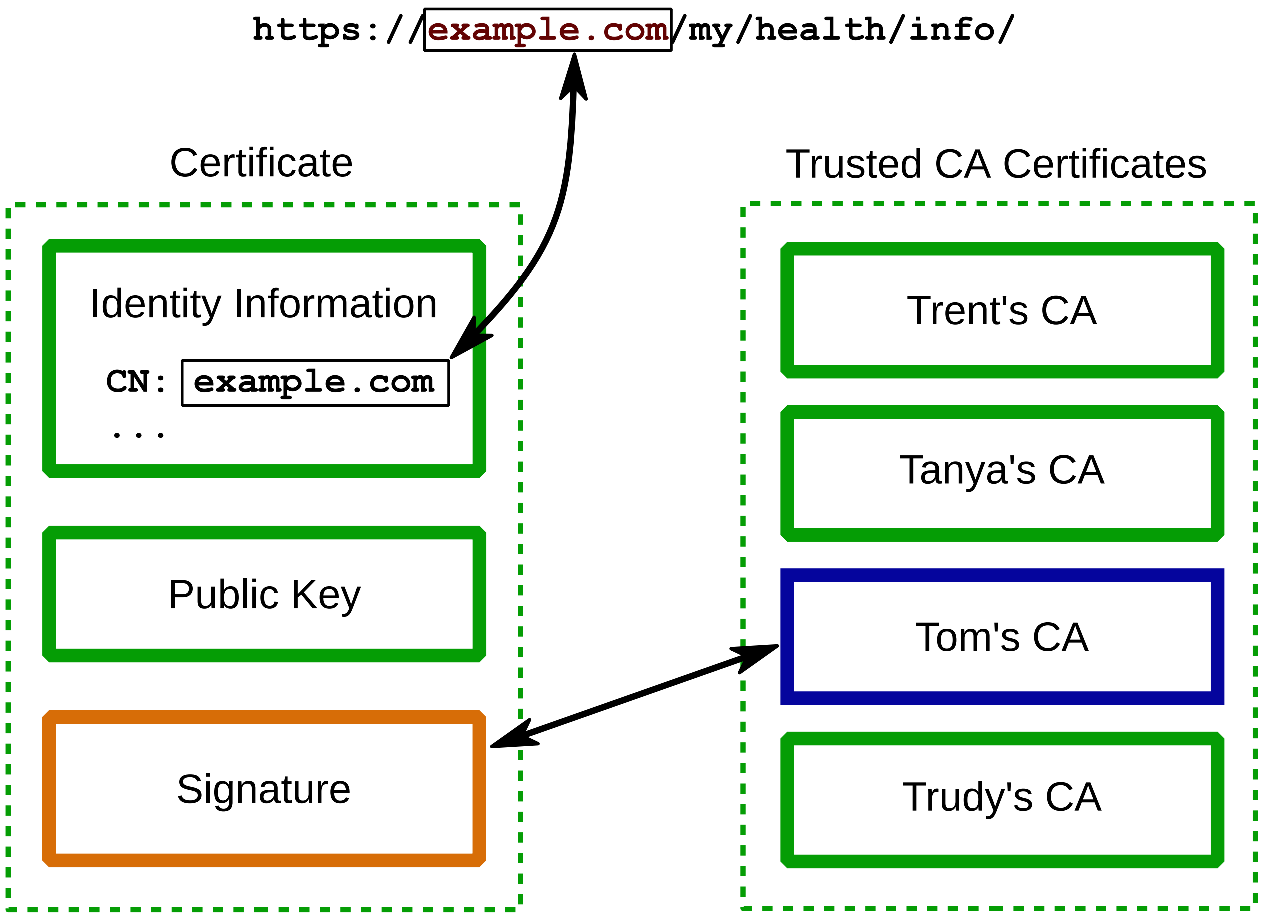
What is a Certificate?



Why Would I Trust a Given Certificate?

- Trust signature of certificate authority (CA)
  - Need CA's public key (the real one!)
  - Need to trust the CA will carefully verify identities
- Verify identity information matches the entity; e.g.:
  - Does the hostname match?
  - Does the email address match?
- Not revoked, not expired, and many other details...

Certificate Validation



Common SSL/TLS Misconceptions

- I didn't validate the certificate, but *some* encryption is better than *none*, right?
  - Nope. In this case, it really is black and white.
  - Faking certificates is the first thing a MitM will try

- I'm not sure if that server supports TLS, so I'll just auto-detect it
  - What happens if the attacker lies about support?
  - Downgrade MitM attack. Often vulnerable:
    SMTP, IMAP, POP3, LDAP, SQL Server TDS, ...

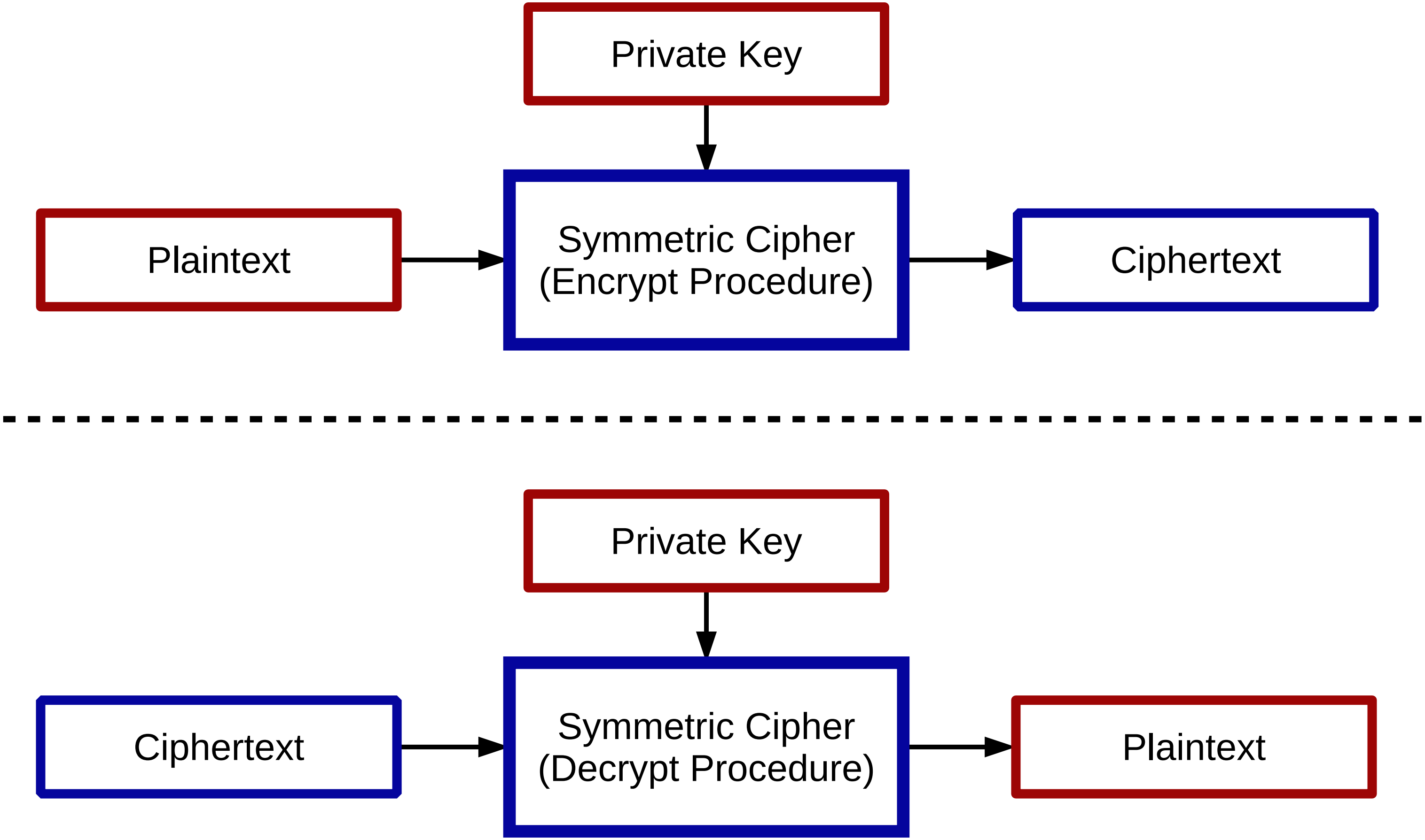
Am I Validating Certificates?

- How can I be sure my implementation is safe?
  - Test it!

- Perform a man-in-the-middle attack on your own communications
  - Set up a MitM service (e.g. Burp, ZAP, socat, ...)
  - Redirect traffic to this service (e.g. `/etc/hosts` file)
  - Ideally, try at least two different invalid certificates
    - Correct hostname, untrusted CA
    - Incorrect hostname, trusted CA

Symmetric Key Refresher

Symmetric Ciphers Illustrated



Stream Ciphers: Encrypting with ⊕

- As it turns out ⊕ offers a simple encryption method:
  - Encrypt with:

    ```
    plaintext ⊕ keystream => ciphertext
    ```

  - Decrypt with:

    ```
    ciphertext ⊕ keystream => plaintext
    ```

- `keystream` bits must be unpredictable and independent
- Works because there is no way for an attacker to know if a guessed keystream is the correct one
- Stream ciphers work by generating high quality pseudorandom keystream based on a secret

Block Ciphers

- Block ciphers are designed to encrypt data only with a specific block size
- When operating on a single block, the following is true:
```
length(plaintext) == length(ciphertext)
```
- The block size is almost always 64 bit (8 bytes) or 128 bit (16 bytes)
- The *key* size does not necessarily equal the block size (e.g. AES-256 has 128 bit blocks)

The Trouble With Block Ciphers

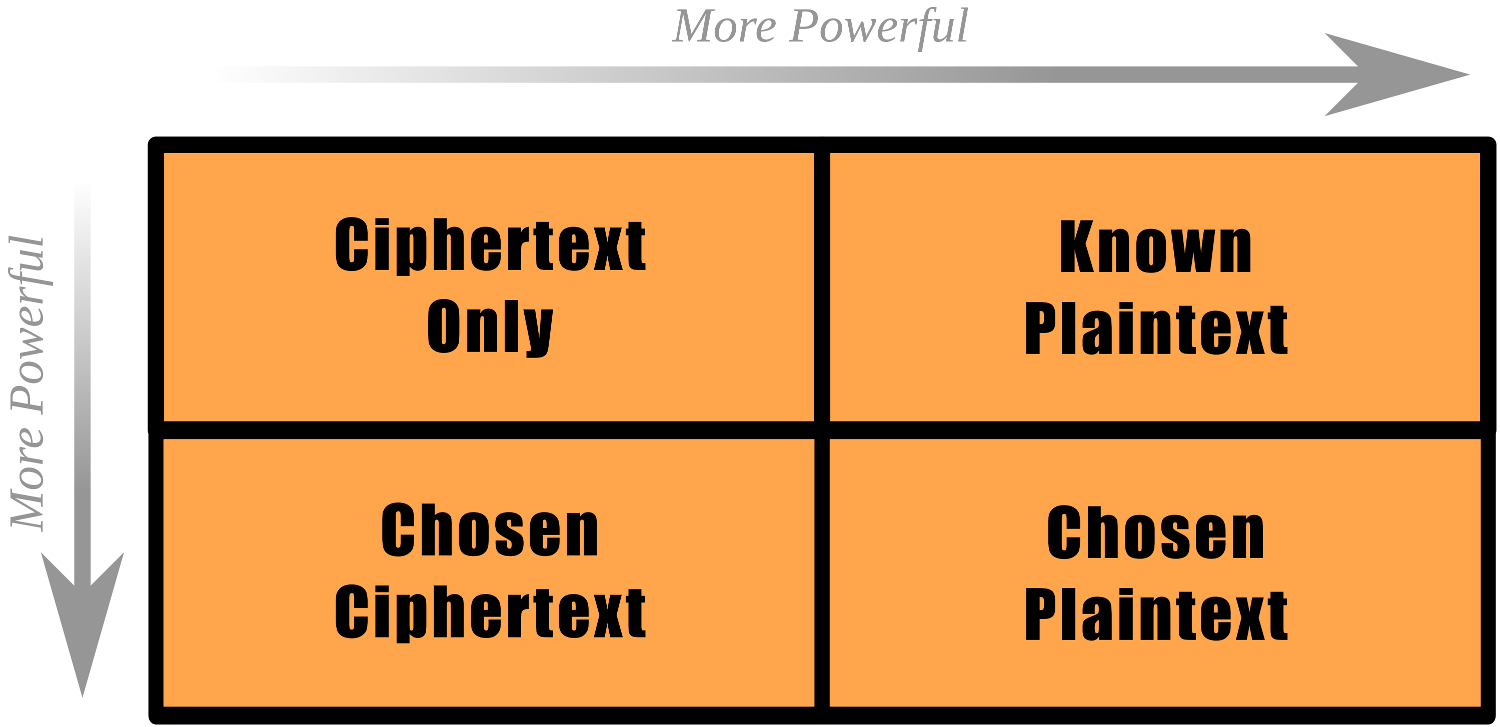
- Suppose my cipher encrypts in 16 byte blocks
- What if I want to encrypt less than 16 bytes?
  - Some kind of block "padding" is needed
- What if I want to encrypt more than 16 bytes?
  - We need to apply a block cipher "mode"

Block Cipher Modes

- Many modes exist to apply block ciphers to long sequences of bits or bytes
- Popular traditional modes: ECB, CBC, OFB, CFB, CTR
  - Traditional cryptography libraries support all of these, and often default to ECB or CBC
  - None of these offer integrity protection
- "Authenticated" modes: XCBC, IACBC, IAPM, OCB, EAX, CWC, CCM, and GCM
  - Built-in integrity protection
  - Many of these are proprietary or patent encumbered
  - Most crypto libraries support only one or two of these

How Cryptographers Got Symmetric Key Wrong

Attack Scenarios



Underestimation of Chosen-Ciphertext

- **Traditional block modes offer no integrity protection**
  - Exposes algorithms to chosen-ciphertext attacks
  - Long-understood issue with stream ciphers and emulating modes (OFB, CFB, CTR, …)
  - Block-swapping attacks on ECB were also obvious
- CBC mode **seems** safer…
  - *But that proved to be false!*

Padding Oracle Attacks

- First credit for discovery to Serge Vaudenay, 2002
  - Warned of attacks on many systems and protocols
  - Did anyone in InfoSec notice? Apparently not.
- Juliano Rizzo and Thai Duong, 2010
  - Major vulnerabilities in .NET, JavaServer Faces, other web frameworks
  - Some allowed for RCE in certain conditions
- Serge told you so! SSL's CBC mode is broken:
  - BEAST (2011)
  - Lucky 13 (2013)
  - POODLE (2014)

Implementing Safe Tokens

"First Generation" APIs

- Typically offer tools for use in several standardized protocols (SSL/TLS, SSH, etc) or formats (PKCS*, X.509, ...)
- Expose low-level primitives for use with custom/proprietary protocols and formats
- Often offer a wide variety of choices in algorithms for compatibility reasons

Problems with Choice

Low-level APIs:

- Don't commit the programmer to particular algorithms
- Don't commit the programmer to specific modes or methods of IV generation
- Don't offer many options for built-in authenticated encryption
- Rarely document the security risks of using the wrong algorithms in the wrong places
- **"By cryptographers for cryptographers"**

Secure Tokens with Low-level API

Programmer decisions in this scenario:

- Select encryption algorithm
- Choose unauthenticated encryption mode
- Securely generate IV
- Embed IV in message
- Select MAC algorithm, apply to all message parts
- Embed MAC in ciphertext envelope
- How to implement parsing, verification, and decryption

Second Generation Crypto Libraries

- Authenticated encryption by default
- Don't come with standards baggage; usable in many contexts
- Make many choices for the programmer, including algorithms and message format
- Examples:
  - KeyCzar
  - NaCl (pronounced "salt")

KeyCzar

- Google/MIT project
- Manages keys locally for easier key rotation
- Integrated IV generation, very simple API
- Uses standard algorithms (AES, RSA, DSA, HMAC/SHA-2)
- Main trunk support for: C++, Java, Python
- Third-party support for: Go, .NET

NaCl

- Created by Daniel J. Bernstein, Tanja Lange, and Peter Schwabe
- Uses high-speed algorithms:
  - Specially optimized elliptic curves (Ed25519)
  - Salsa20 and Poly1305 for authenticated encryption
  - Option to use slower, more standard algorithms
- libsodium is a more portable clone of NaCl
  - Support for: C, C++, Common Lisp, Erlang, Haskell, Java, Lua, .NET, NodeJS, Objective C, PHP, Python, Ruby, ...
  - Includes scrypt (by Colin Percival)

KeyCzar/libsodium Comparison

KeyCzar Pros:

- Automatic IV generation
- Standard algorithms by default
- Key management

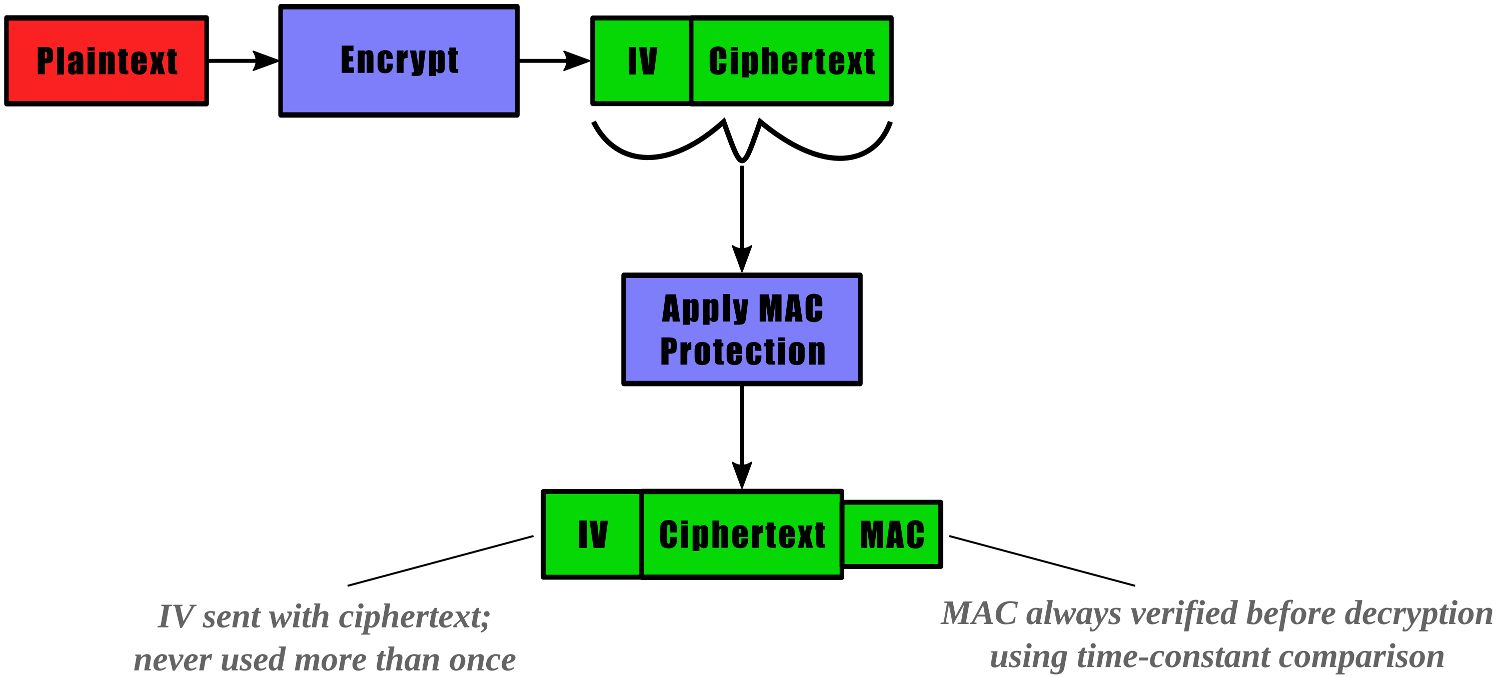
libsodium Pros:

- Faster algorithms by default (good for embedded)
- Preliminary scrypt support (C API)
- More widely supported in high-level languages

If You MUST Do It Yourself...

- Consider using GCM mode
  - Very fast with integrated integrity protection
  - Pitfall: Be sure to never reuse an IV!!!
  - Pitfall: May be vulnerable to timing attacks
- ... Or use explicit integrity protection
  - Slower, more error-prone
  - Use AES-CBC and random IV, **both** HMAC protected
  - Pitfall: Be sure to check HMAC first during decryption
  - Pitfall: Use time-constant HMAC verification!
- Paranoid? Use AES-GCM with an HMAC

Explicit Integrity Protection

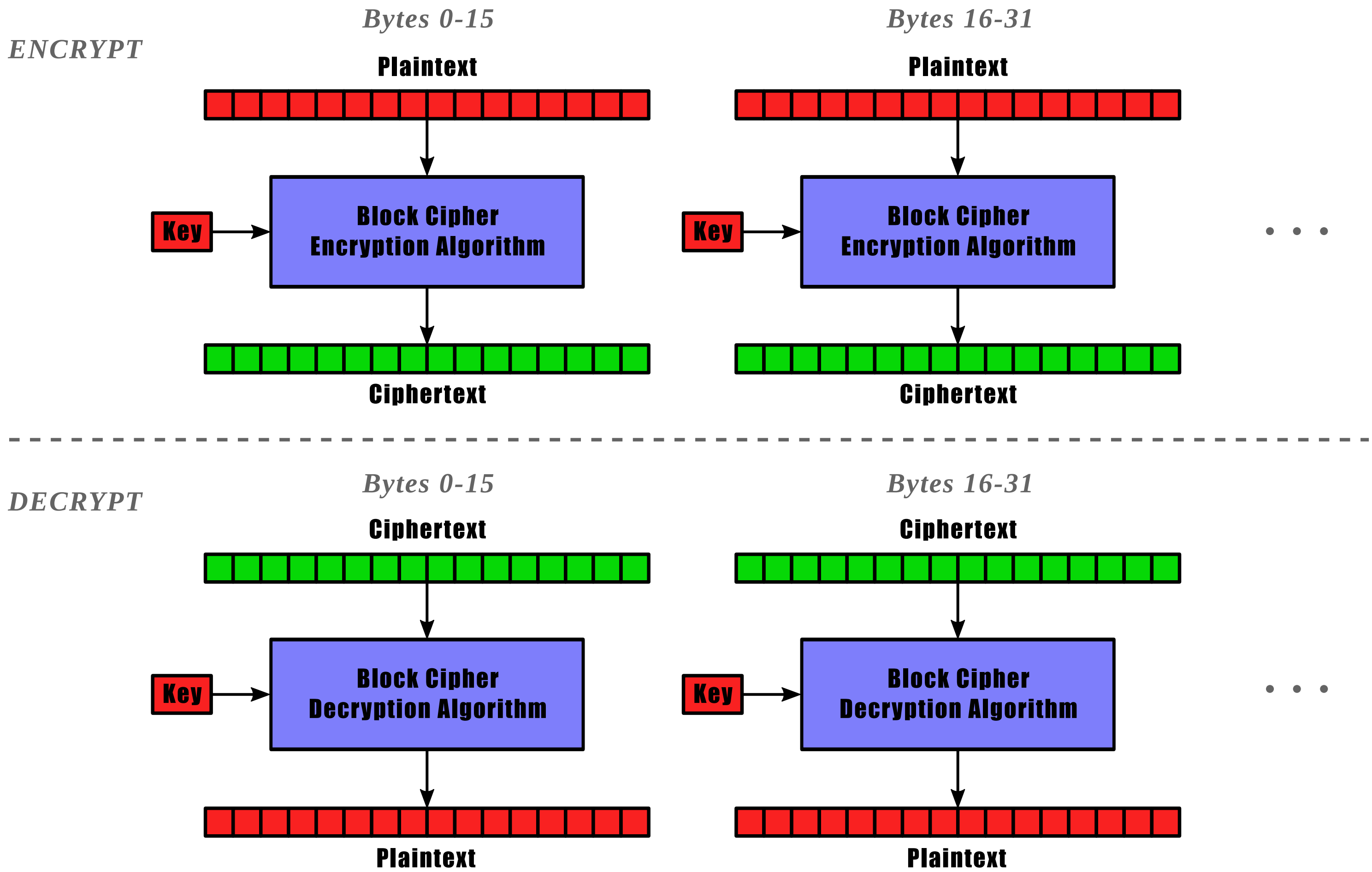


Further Reading

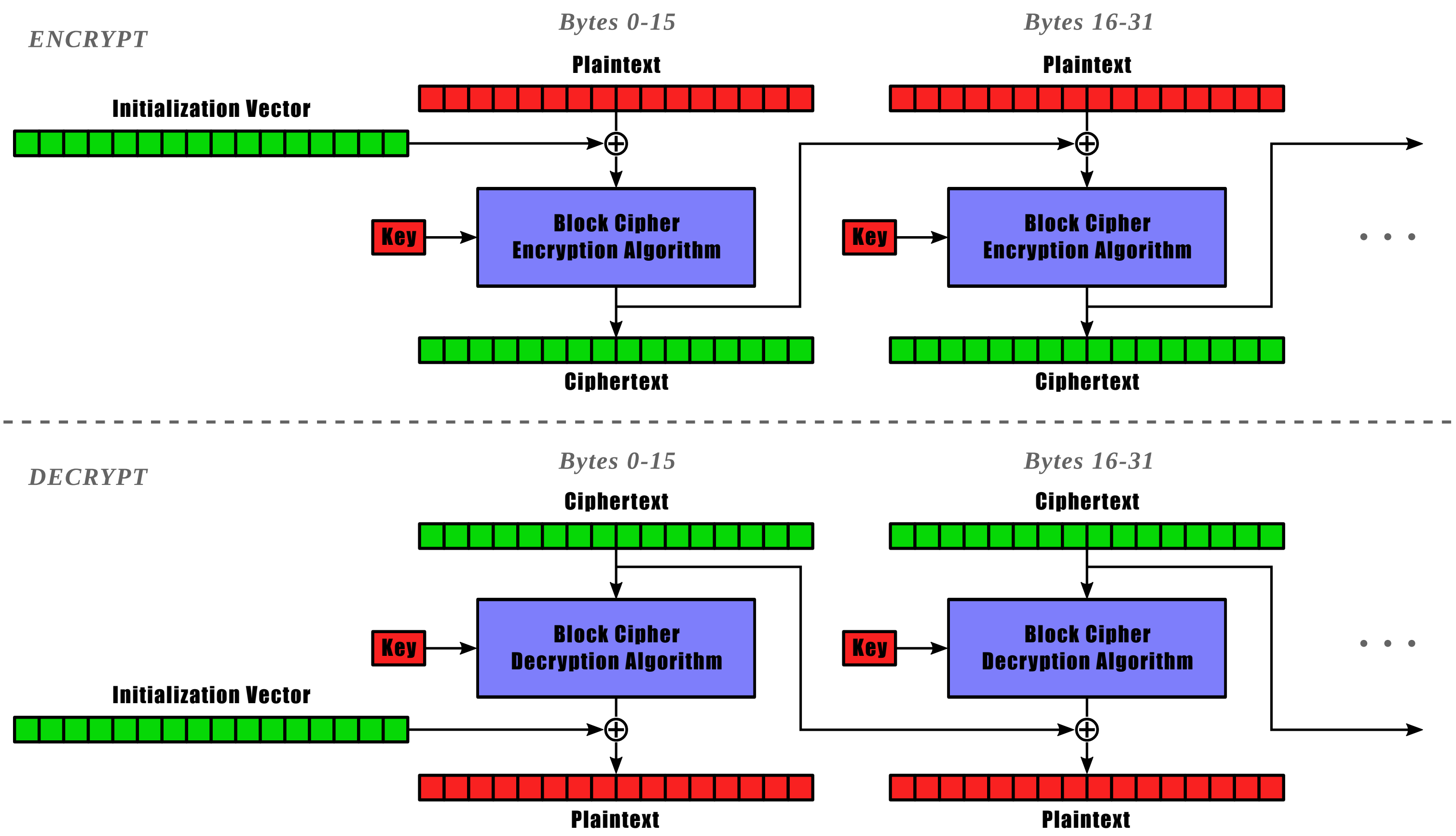
- Let's Encrypt
- Bletchley
- Cryptopals Crypto Challenges
- Padding Oracles Everywhere

Thank You!

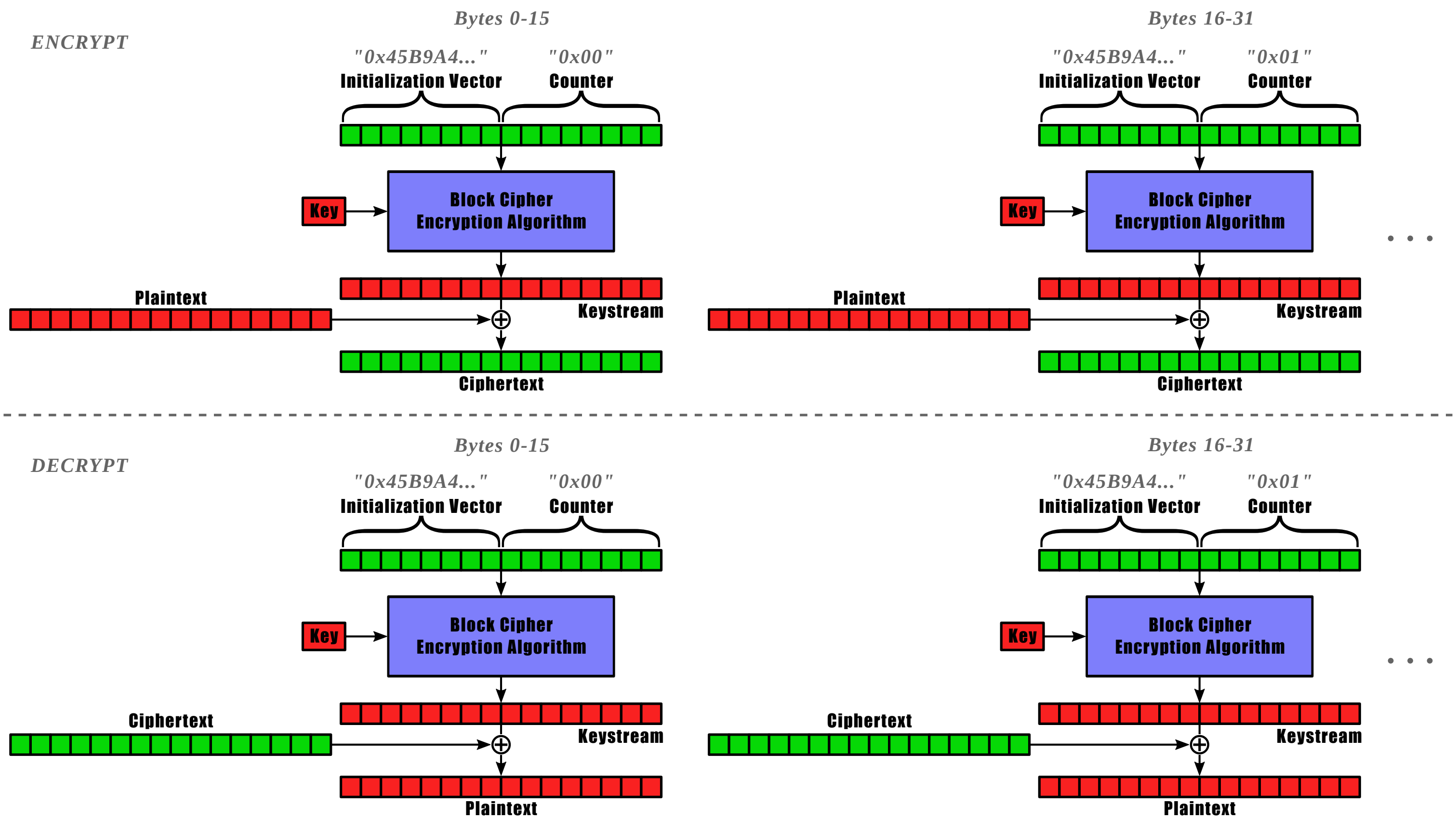
ECB Mode


ENCRYPT
Bytes 0-15
Plaintext
Key
Block Cipher
Encryption Algorithm
Ciphertext
Bytes 16-31
Plaintext
Key
Block Cipher
Encryption Algorithm
Ciphertext
. . .
DECRYPT
Bytes 0-15
Ciphertext
Key
Block Cipher
Decryption Algorithm
Plaintext
Bytes 16-31
Ciphertext
Key
Block Cipher
Decryption Algorithm
Plaintext
. . .

CBC Mode


ENCRYPT
Bytes 0-15
Plaintext
Bytes 16-31
Plaintext
Initialization Vector
Key
Block Cipher
Encryption Algorithm
Key
Block Cipher
Encryption Algorithm
Ciphertext
Ciphertext
DECRYPT
Bytes 0-15
Ciphertext
Bytes 16-31
Ciphertext
Key
Block Cipher
Decryption Algorithm
Key
Block Cipher
Decryption Algorithm
Initialization Vector
Plaintext
Plaintext

CTR Mode



ENCRYPT
Bytes 0-15
"0x45B9A4..."
Initialization Vector
"0x00"
Counter
Key
Block Cipher Encryption Algorithm
Plaintext
Keystream
Ciphertext
Bytes 16-31
"0x45B9A4..."
Initialization Vector
"0x01"
Counter
Key
Block Cipher Encryption Algorithm
Plaintext
Keystream
Ciphertext
DECRYPT
Bytes 0-15
"0x45B9A4..."
Initialization Vector
"0x00"
Counter
Key
Block Cipher Encryption Algorithm
Ciphertext
Keystream
Plaintext
Bytes 16-31
"0x45B9A4..."
Initialization Vector
"0x01"
Counter
Key
Block Cipher Encryption Algorithm
Ciphertext
Keystream
Plaintext