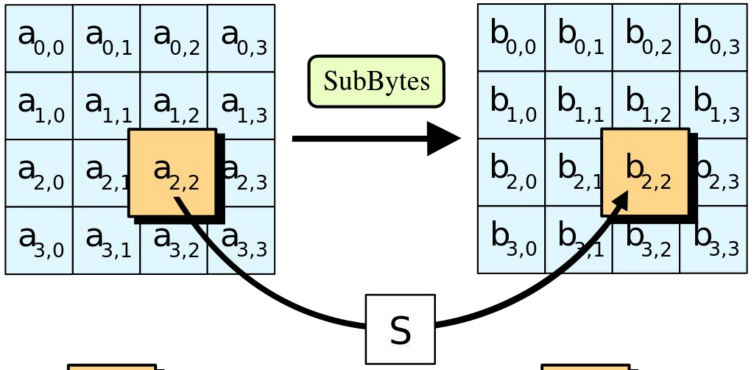
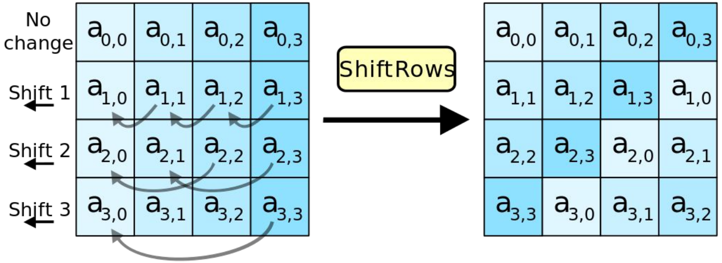
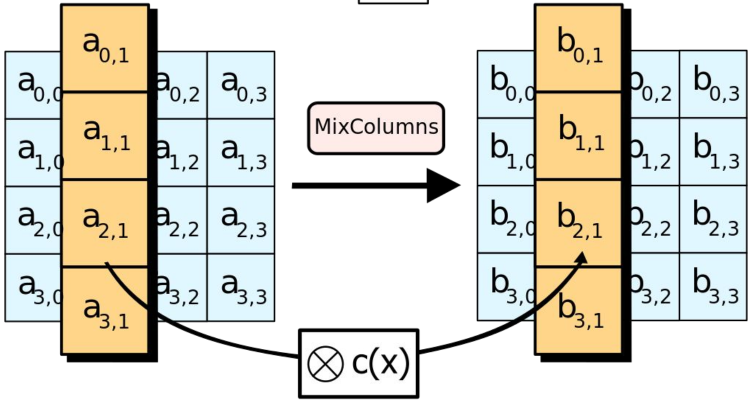
# Next Directions in Cryptography

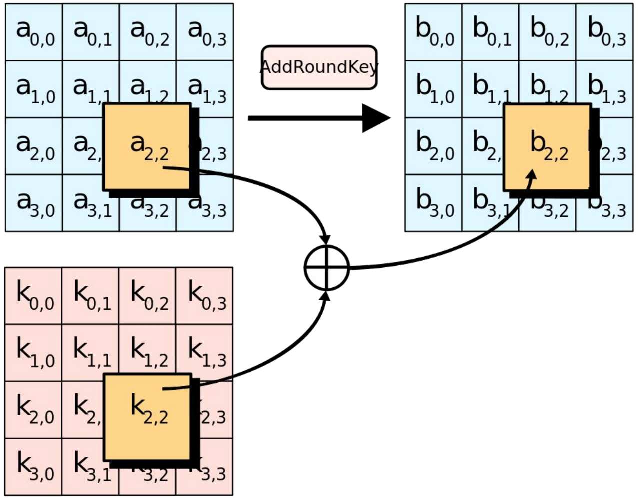
# Symmetric-Key Encryption

AES (Advanced Encryption Standard) provides confidentiality

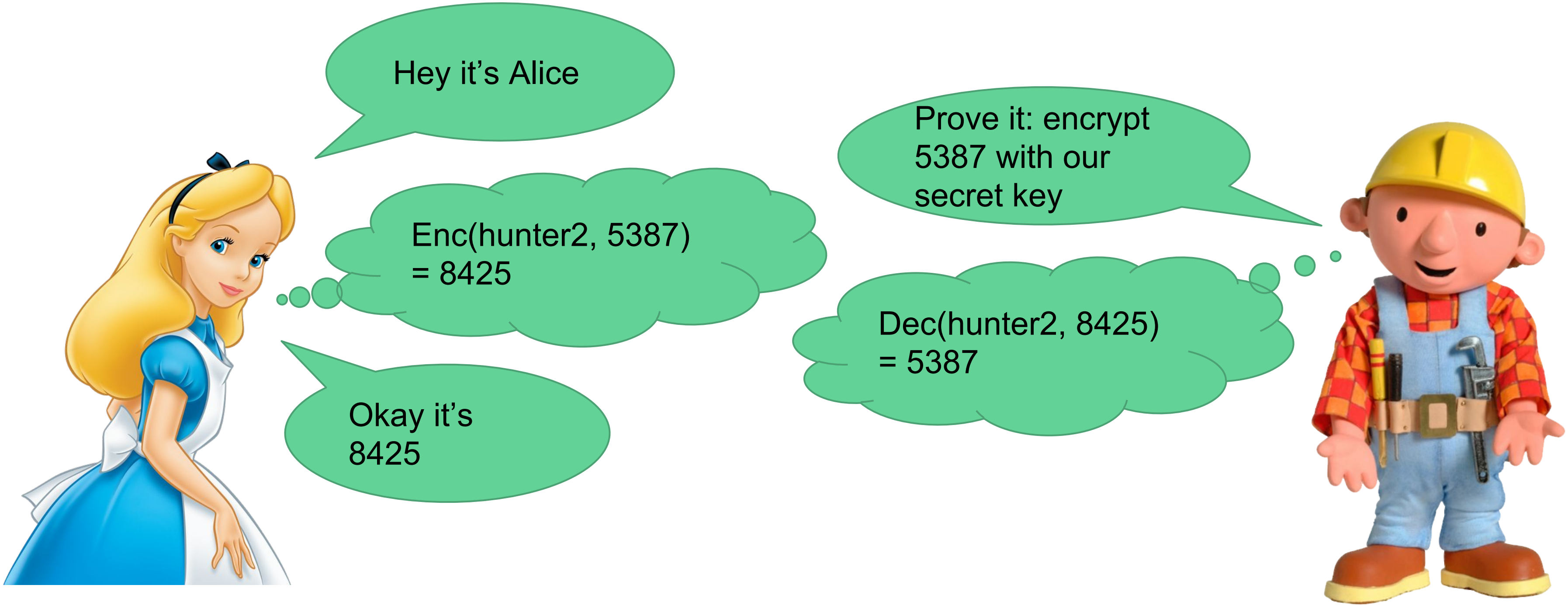


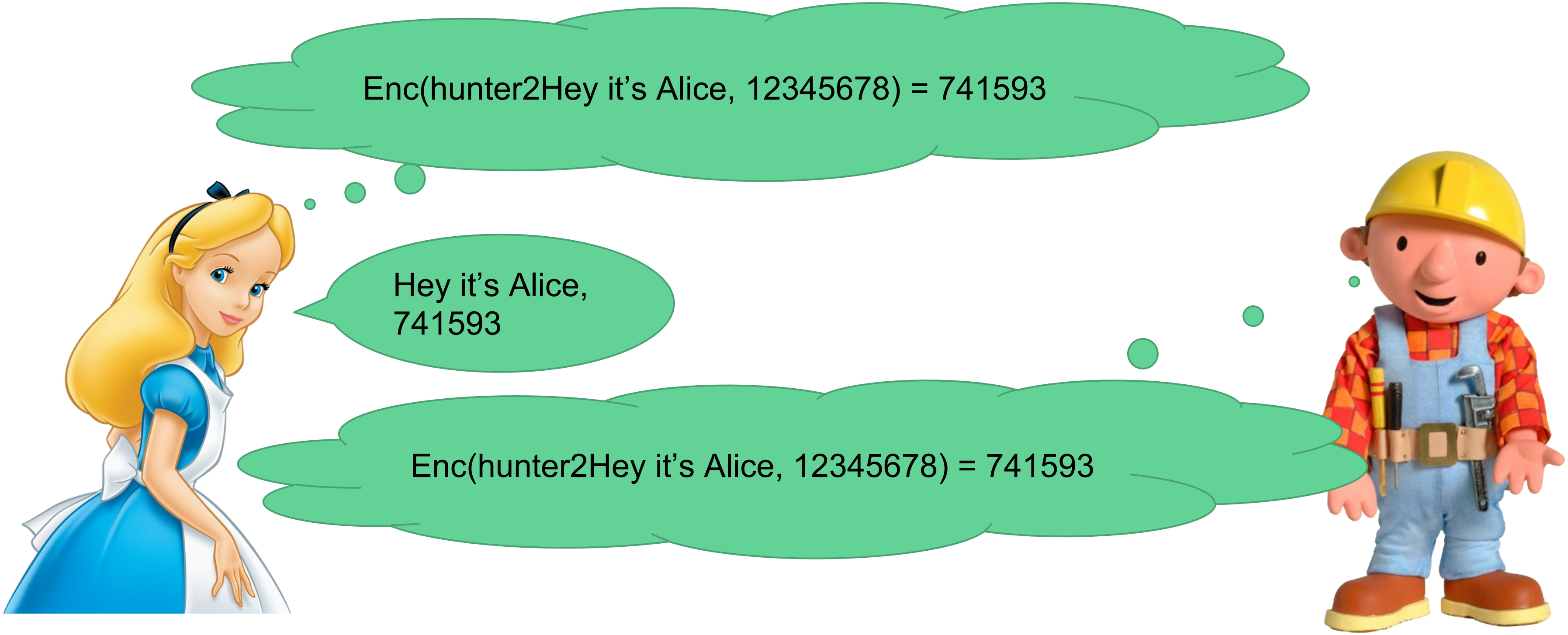



# Challenge-Response Authentication



# Message-Authentication Codes (MACs)



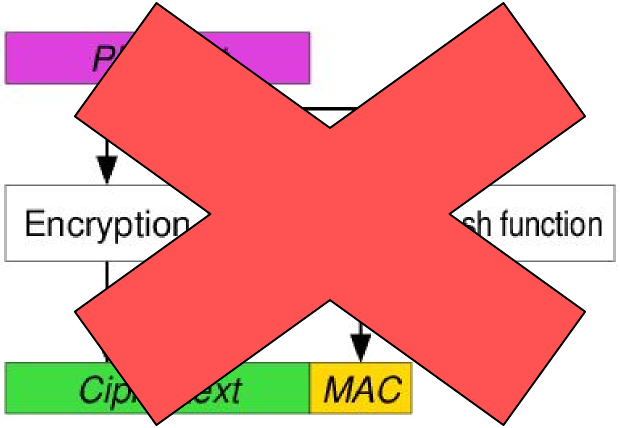
# Hash-based MACs

SHA256(hunter2Hey it's Alice) = 741593

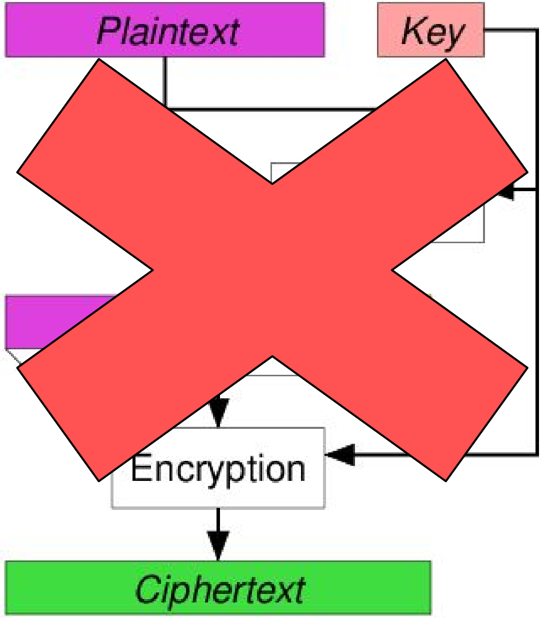
Hey it's Alice,
741593

SHA256(hunter2Hey it's Alice) = 741593

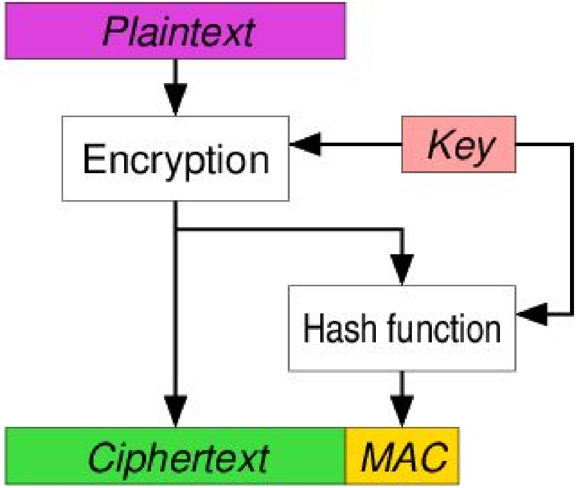
# Composing Authentication and Encryption




Encrypt and MAC




MAC then Encrypt




Encrypt then MAC

# Dedicated Authenticated-Encryption Schemes

- Can we achieve privacy and integrity with a single primitive?
  - Network communications always need authentication
  - Easier to implement a single primitive than two primitives
  - Faster than composing encryption and MACs
- Yes! AES-GCM
  - Encrypt the message with AES
  - Interpret the message and key as polynomials, multiply and mod by $x^{128}+x^7+x^2+x+1$

# Competition for Authenticated Encryption: Security, Applicability, and Robustness

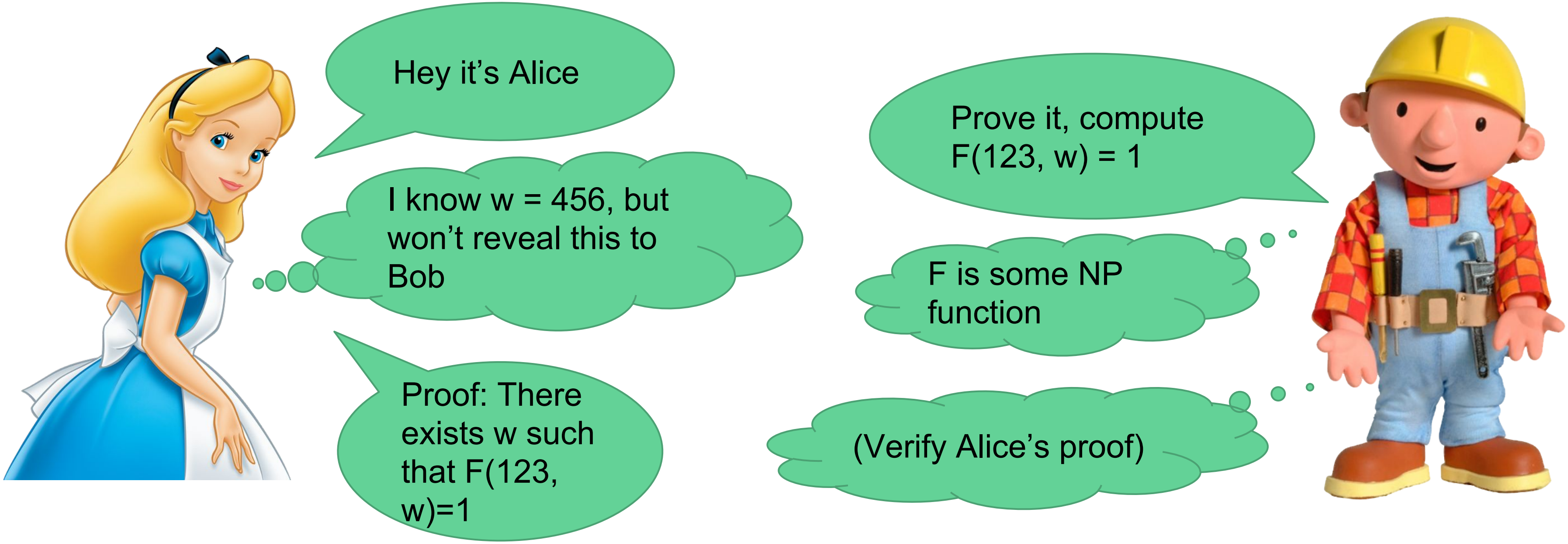
“CAESAR will identify a portfolio of authenticated ciphers that

1. offer advantages over AES-GCM
2. are suitable for widespread adoption”

http://competitions.cr.yp.to/caesar-submissions.html

# zk-SNARKs

Non-interactive zero-knowledge proofs allow us to verify we know some secret value without revealing it (zero-knowledge) by performing some computation



# zk-SNARKs

Non-interactive zero-knowledge proofs allow us to verify we know some secret value without revealing it (zero-knowledge) by performing some computation

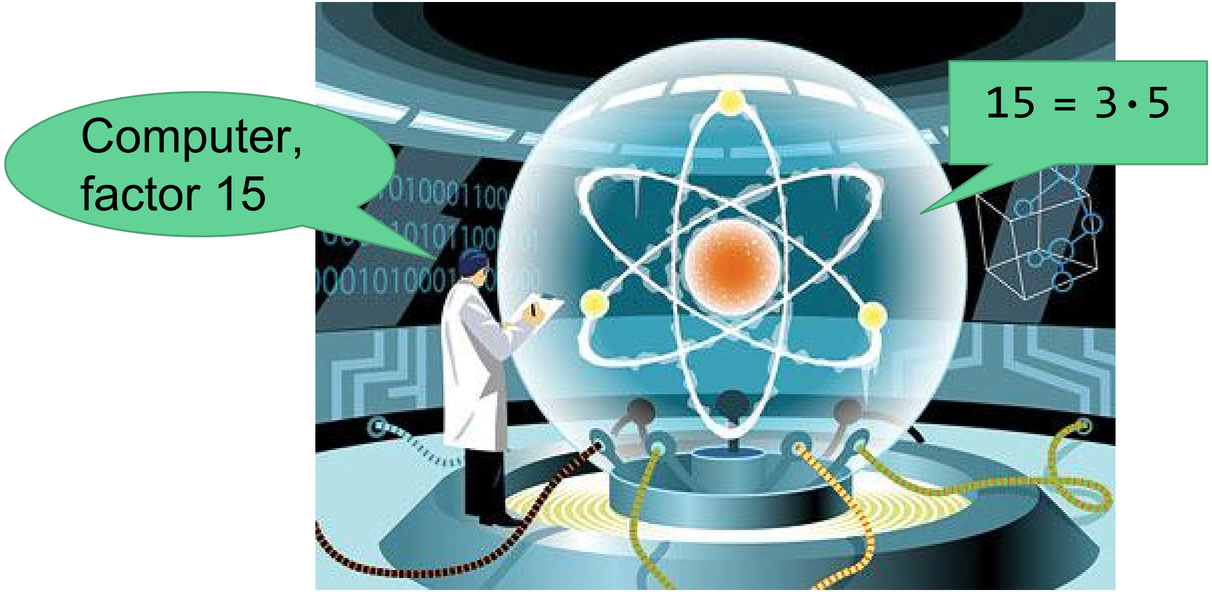
We desire:

- **Zero Knowledge** - Bob only knows the statement is true, not Alice's private *w*
- **Succinctness** - Proof is short and easy/quick to verify
- **Non-interactivity** - No back and forth interaction, only a single proof from Alice
- **Proof of Knowledge** - Bob verifies both the statement AND that Alice has a *w*

Example of use: Zcash (https://z.cash)

# Attacking Public-key Crypto

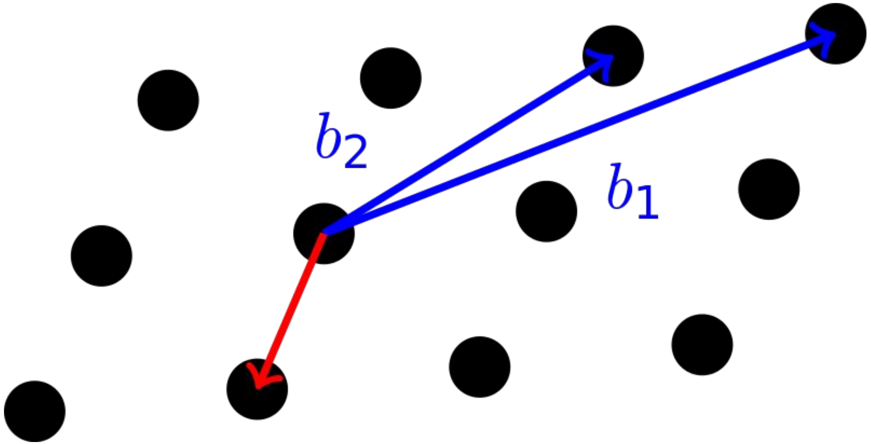
- Diffie-Hellman Key Exchange: given $g^x$, find $x$
- RSA Encryption / Signatures: given $n = p \cdot q$, find $p$ and $q$



# Post-Quantum Cryptography

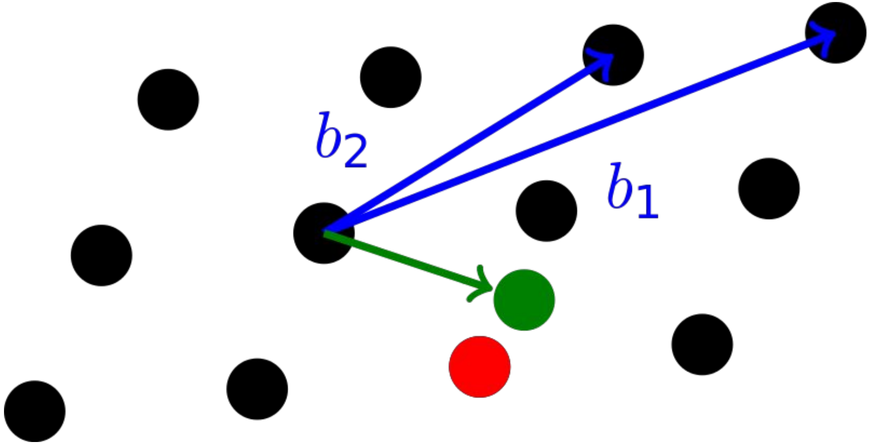
- Many schemes resist attacks from quantum computers
  - Secret-key cryptography
  - Lattice-based cryptography

# PQC: problems from lattices




Shortest Vector Problem

Closest Vector Problem



# Post-Quantum Cryptography

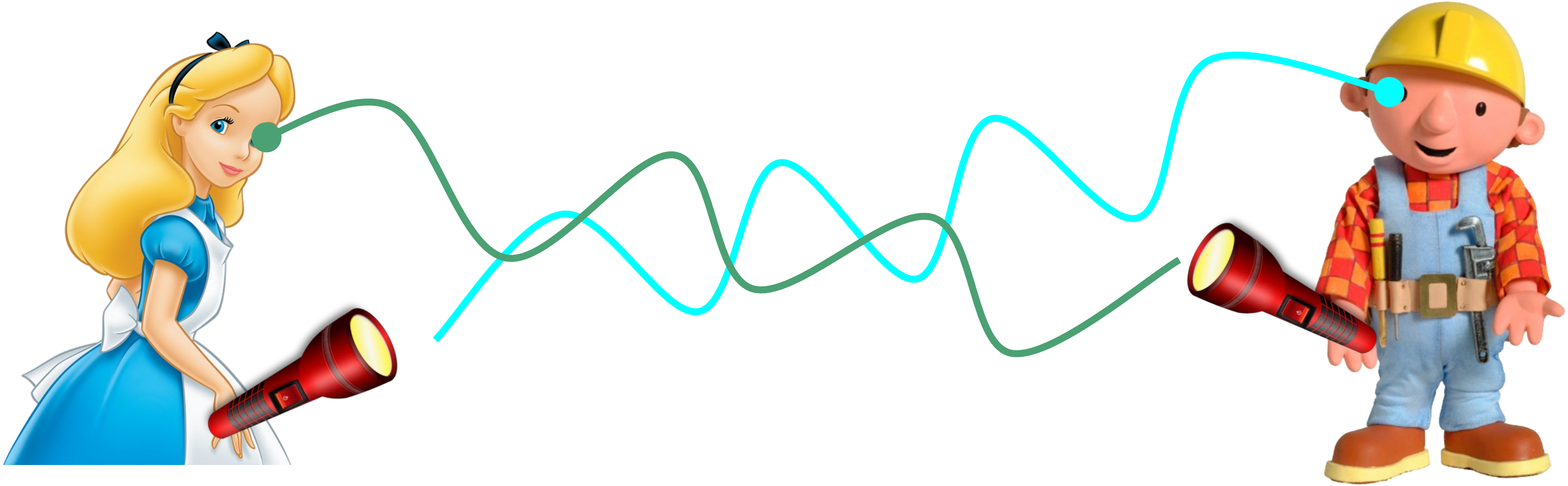
- Many schemes resist attacks from quantum computers
  - Secret-key cryptography
  - Lattice-based cryptography
  - Hash-based cryptography
  - Code-based cryptography
  - Multivariate-quadratic-equations cryptography
  - Meet-privately-in-a-sealed-vault cryptography
- Why don’t we use them?
  - Efficiency
  - Confidence
  - Usability

# NIST PQC

- The National Institute of Standards and Technology (NIST) is looking to standardize quantum-resistant public-key crypto schemes
- Evaluation criteria
  - Security
  - Cost
    - Key, ciphertext, signature sizes
    - Computational efficiency
  - Simplicity
- Timeline
  - Submit your proposal by November 30
  - 3–5 years of public scrutiny
  - 2 years of writing standards

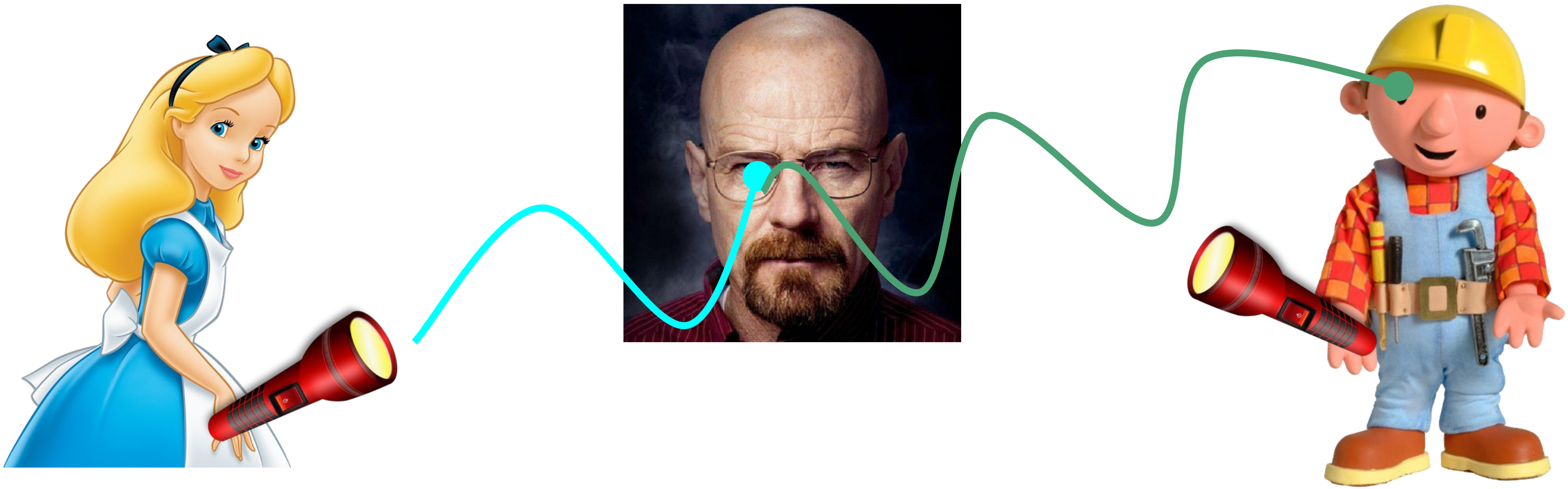
# Quantum Cryptography

- Quantum key distribution
- Quantum signing tokens



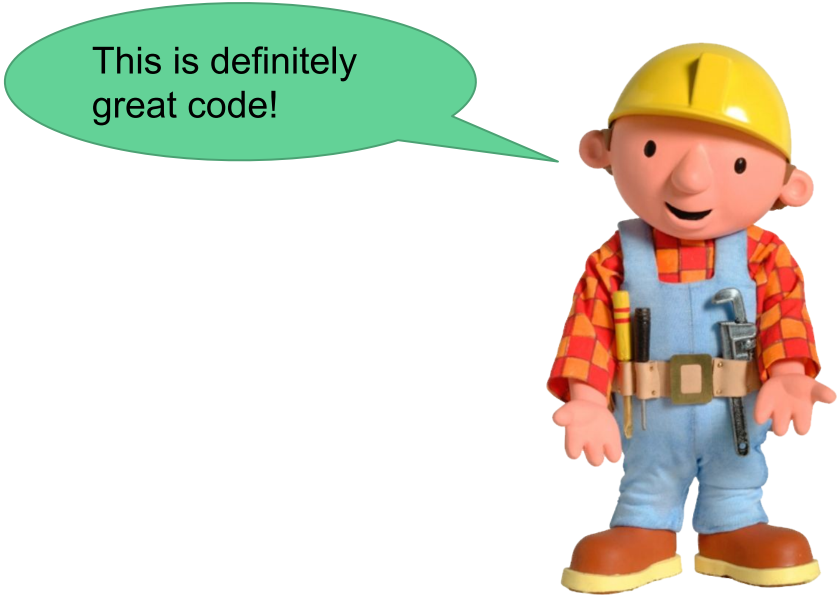
# Quantum Cryptography

- Observing a quantum state irreversibly changes it



# Timing Attacks

```
def secureCompare(one, two):
    if len(one) != len(two):
        return False
    x = 0
    while x < len(one):
        if one[x] != two[x]:
            return False
        x = x + 1
    return True
```



# Timing Attacks

```
def secureCompare(one, two):
    if len(one) != len(two):
        return False
    x = 0
    while x < len(one):
        if one[x] != two[x]:
            return False
        x = x + 1
    return True
```

Code timing depends on *value* of secret data

Reduces bruteforce from exponential time to linear (w.r.t. length)

# Brumley's RSA Timing Attacks

```
# Basic square-and-multiply
def square_and_multiply(c, d, n):
    x = c
    while d:
        x = mod(x * x, n)
        if d & 1:
            x = mod(x * c, n)
        d >>= 1
    return x
```

To compute $c^d$ mod N, OpenSSL RSA used an optimized version of this idea

Timing issue was exposed depending on whether a given bit was 1 or 0

# Brumley's RSA Timing Attacks

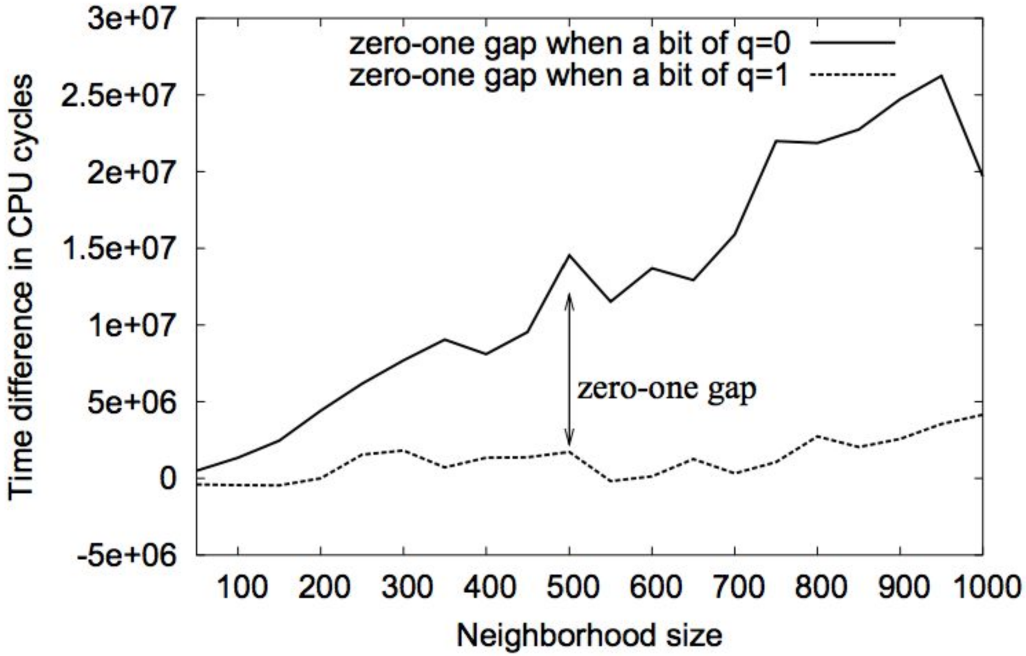
Hey it's Alice

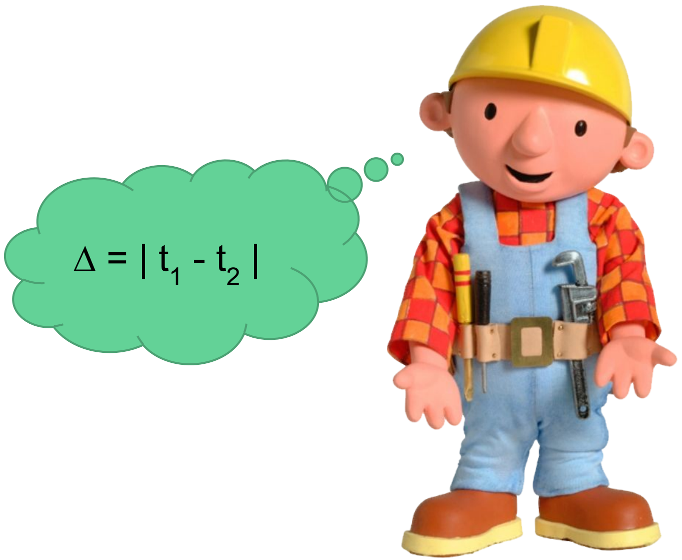
Ok… <time passes>

Please decrypt
($g*R^{-1}$ mod N) and
($g_{hi}*R^{-1}$ mod N)

$\Delta = | t_1 - t_2 |$

# Brumley’s RSA Timing Attacks



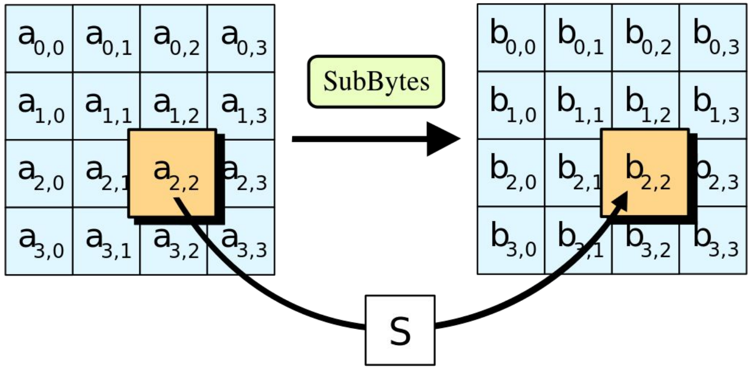
# Cache timing attacks

```
def secureLookup(one):
    x = 0
    r = 0
    while x < len(one):
        r += table[42 * one[x]]
    return r
```

Even if your code runs with constant time, what about *memory access patterns*?

Perfect, right?

# AES cache attack




Local: k[0] ^ 43 is slowest (for k[0] = 0)

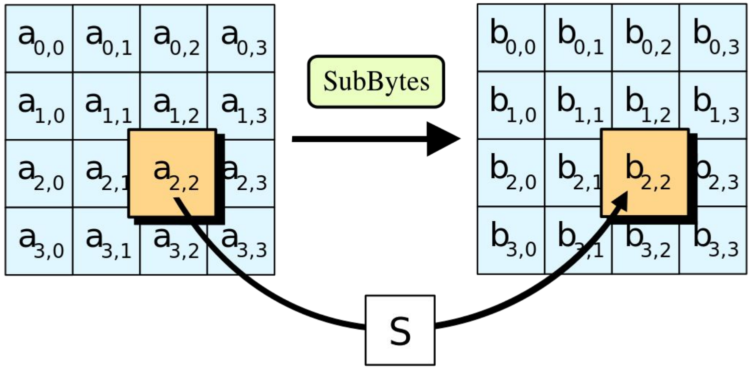
Victim: k[0] ^ 203 is slowest

-- First byte of key is 43 ^ 203

AES implementations use lookup tables during cipher rounds for performance

**djb attack:** Get same CPU and AES implementation as victim and measure timings -- whichever

# AES cache attack




PRIME+PROBE:

1. Fill up cache with known data
2. Run victim AES process
3. See which part of your cache was removed
4. You know which part of the lookup table was accessed!

# Homework

- Find a cool topic in cryptography from the past few years and write up a short summary
  - E.g. browse https://arxiv.org/list/cs.CR/recent for a paper with a title you somewhat understand
- Send an email with your summary to cm7bv@virginia.edu with the subject “MST Assignment 8 - <YOUR_UVA_ID>”
- **Don’t hesitate to ask questions!**

# Additional Reading

- How to choose an Authenticated Encryption mode by Matt Green (https://blog.cryptographyengineering.com/2012/05/19/how-to-choose-authenticated-encryption/)
- Introduction to post-quantum cryptography by Daniel J. Bernstein (https://pqcrypto.org/www.springer.com/cda/content/document/cda_downloaddocument/9783540887010-c1.pdf)
- Timing attacks and good coding practices (http://crypto.stackexchange.com/questions/41691/timing-attack-and-good-coding-practices)