Introduction to Crypto (CP2)

Jacob Stolker
University of Illinois
CS 461

MP3 - Cryptography

- Checkpoint 1 (20 points)
- Checkpoint 2 (80 points)
- Important notes:
  - Almost no partial credit, either it works or it does not.
  - We recommend to always test your solutions.
  - Some problems in Checkpoint 2 require long computation times, so we recommend to get started early.
  - Run on Python 3.10 or below if outside VM (VM should be 3.6 and should have no problems)
  - Read the Docs carefully

Hash Functions (revisited)

- Map arbitrarily long input strings to a fixed-length output
- Not all hash functions are cryptographically useful!
- A cryptographic hash should be:
    - Preimage Resistant (One-Way): Given H(a) it's hard to find a
    - Collision Resistant: It's hard to find (a,b) s.t. H(a) == H(b)
        - We will break this property in 3.2.2!
    - Second Preimage Resistant: Given a, it's hard to find b s.t. H(a) == H(b)
        - We broke this property in 3.1.6!

3.2.2 – MD5 Collisions

- Goal: 2 programs with the same MD5 hash, but different behavior
- MD5 does not have strong collision resistance
- Fastcoll lets you generate Chosen Prefix Collisions

3.2.2 – MD5 Collisions

- Goal: 2 programs with the same MD5 hash, but different behavior
- MD5 does not have strong collision resistance
- Fastcoll lets you generate Chosen Prefix Collisions

```
#!/usr/bin/env python3
# -*- coding: latin-1 -*-
blob = """
<<SOMETHING GENERATED BY FASTCOLL>>
```

```
#!/usr/bin/env python3
# -*- coding: latin-1 -*-
blob = """
<<DIFFERENT BYTES GENERATED BY
FASTCOLL>>
```

3.2.2 – MD5 Collisions

- Goal: 2 programs with the same MD5 hash, but different behavior
- MD5 does not have strong collision resistance
- Fastcoll lets you generate Chosen Prefix Collisions

```
#!/usr/bin/env python3
# -*- coding: latin-1 -*-
blob = """
<<SOMETHING GENERATED BY FASTCOLL>>
"""
from hashlib import sha256
print(sha256(blob).hexdigest())
```

```
#!/usr/bin/env python3
# -*- coding: latin-1 -*-
blob = """
<<DIFFERENT BYTES GENERATED BY
FASTCOLL>>
"""
from hashlib import sha256
print(sha256(blob).hexdigest())
```

3.2.2 – MD5 Collisions - Tips

- Will not work on > Python 3.10
- Make sure prefix file has a new-line

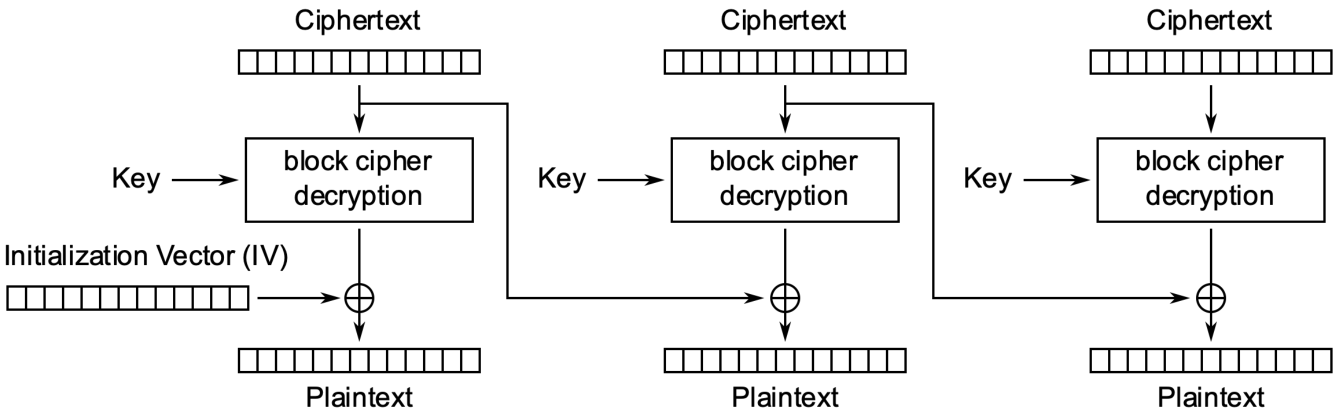
```
#!/usr/bin/env python3
# -*- coding: latin-1 -*-
blob = """
<<SOMETHING GENERATED BY FASTCOLL>>
"""
from hashlib import sha256
print(sha256(blob).hexdigest())
```

```
#!/usr/bin/env python3
# -*- coding: latin-1 -*-
blob = """
<<DIFFERENT BYTES GENERATED BY
FASTCOLL>>
"""
from hashlib import sha256
print(sha256(blob).hexdigest())
```

3.2.3 – Padding Oracle to Decrypt AES

- We have a Padding Oracle Server (IP in docs) that has an expected plaintext
  1. Receive IV and an ciphertext
  2. Decrypt the ciphertext with IV and its secret key
  3. Compare the decrypted message with expected plaintext
     - Error 500: Padding Scheme is invalid
     - Error 404: Decrypted Message =/= Expected Plaintext (Padding scheme is correct)
     - Success : Decrypted Message == Expected Plaintext
- Using the server response, and many Ciphertexts, you can figure out the plaintext!

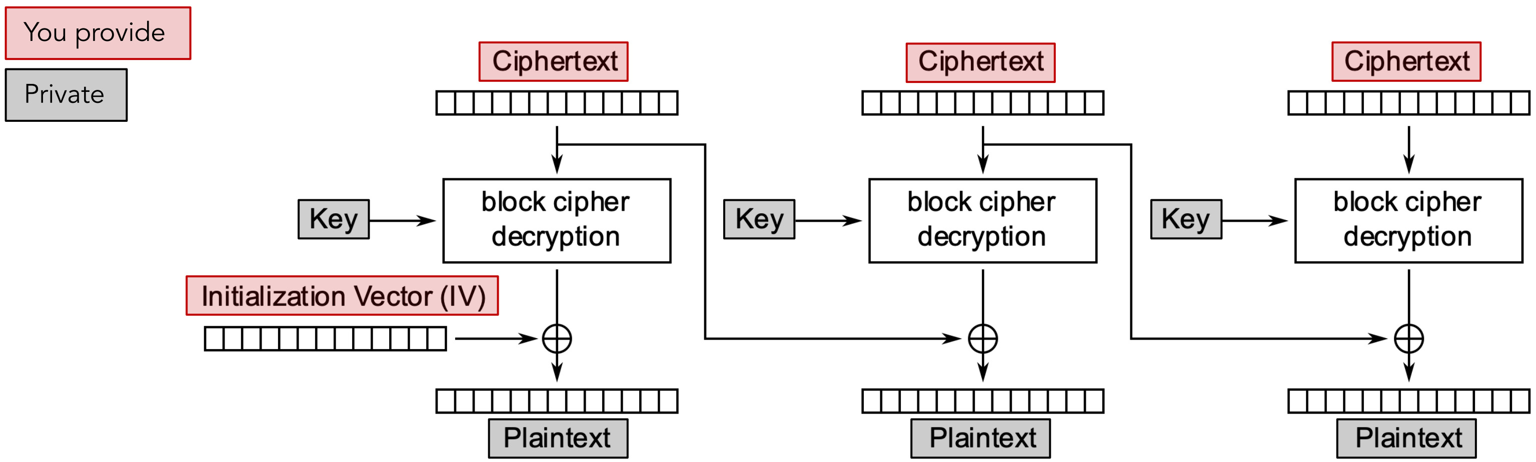
3.2.3 – Padding Oracle to Decrypt AES



Cipher Block Chaining (CBC) mode decryption

The symbol ⊕ means "XOR"

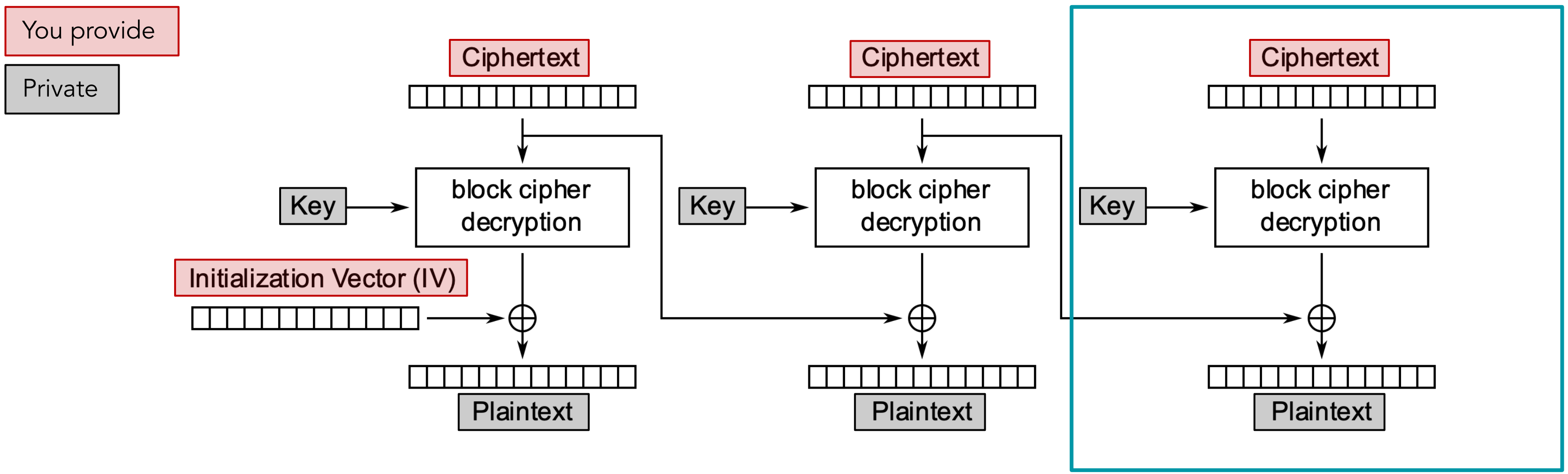
3.2.3 – Padding Oracle to Decrypt AES



The symbol ⊕ means "XOR"

1. If the plaintext does not end with valid padding (\x10\x0f…), error 500
2. If the plaintext doesn't match the expected string, error 404
3. If the plaintext matches what the server was expecting, success

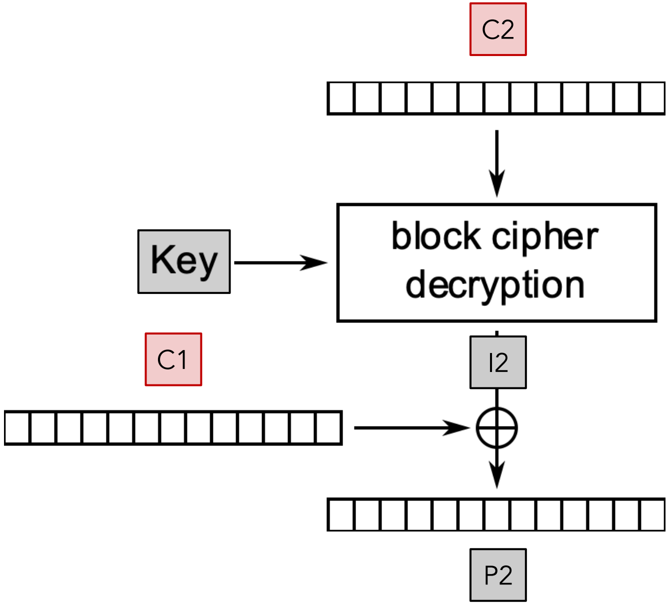
3.2.3 – Padding Oracle to Decrypt AES



The symbol ⊕ means "XOR"

We build out our solution in reverse

3.2.3 – Padding Oracle to Decrypt AES



Notation:
CX_z denotes the zth byte of Cipherblock X

We control C1 and C2 to "probe" the server and see what it responds

3.2.3 – Padding Oracle to Decrypt AES

This is the original plaintext:

P2	0x20	E	C	E	4	2	2	0x20	S	E	C	R	E	T	0x10	0x0f

Padding Bytes

3.2.3 – Padding Oracle to Decrypt AES

What if the last byte is 0x10?

P2′	0x20	E	C	E	4	2	2	0x20	S	E	C	R	E	T	0x10	0x10

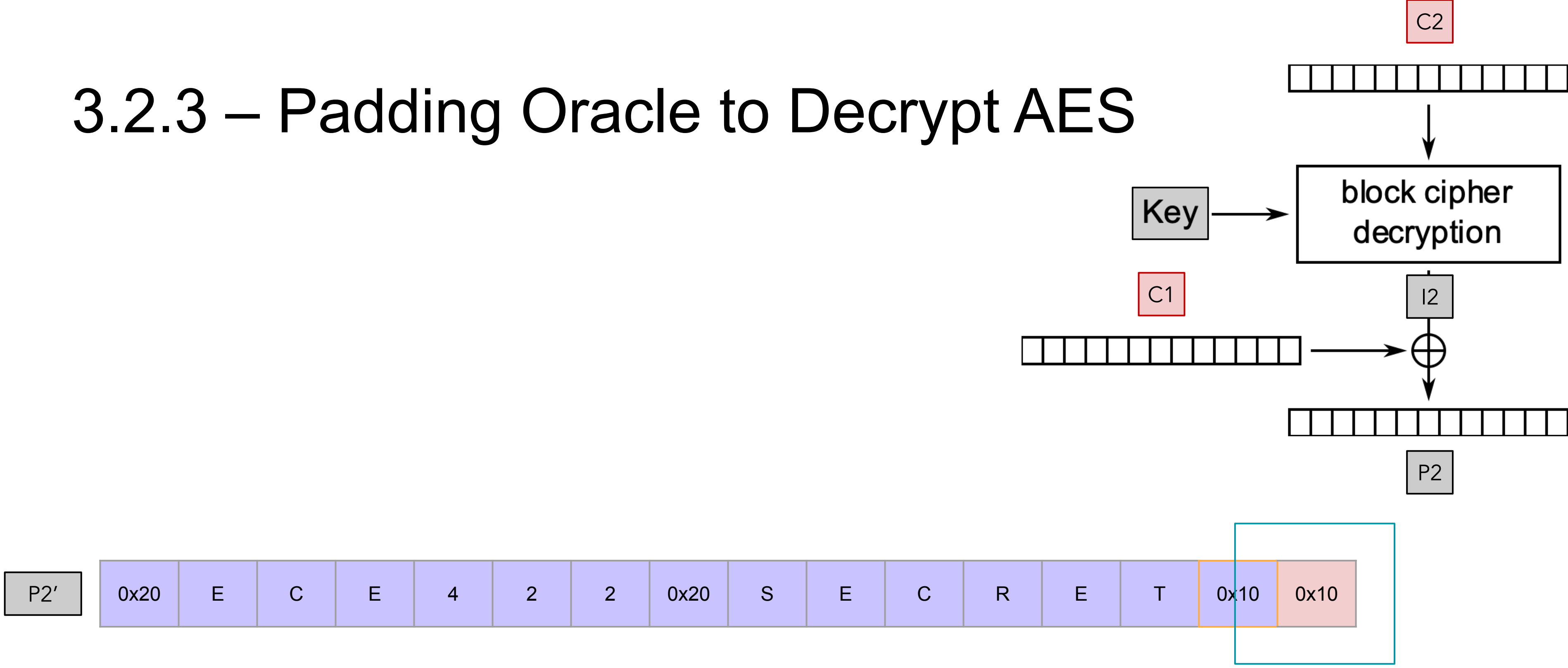
3.2.3 – Padding Oracle to Decrypt AES

Interpreted by server as:

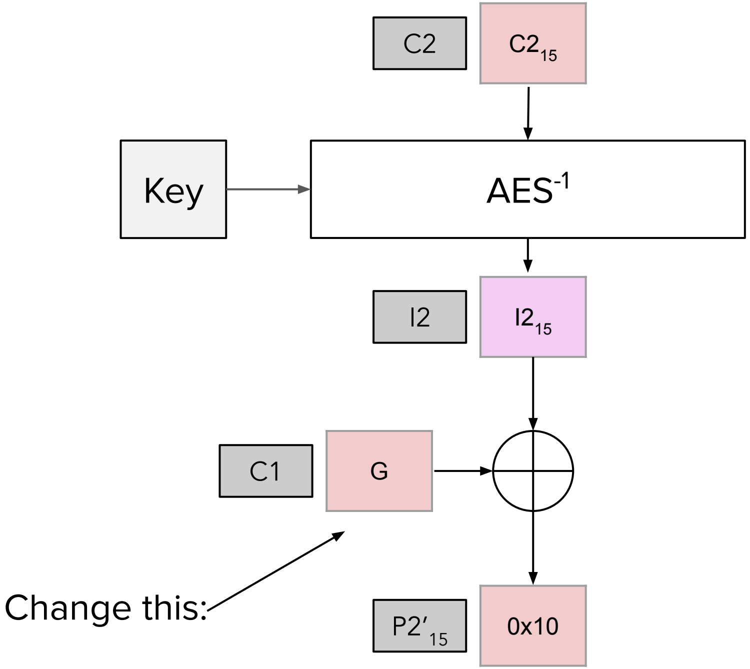
P2′	0x20	E	C	E	4	2	2	0x20	S	E	C	R	E	T	0x10	0x10

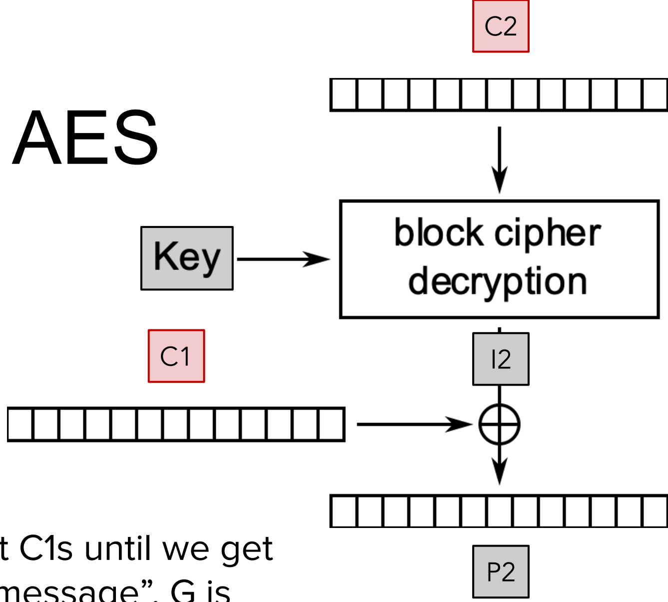
Server: “Valid Padding, incorrect message”!!

3.2.3 – Padding Oracle to Decrypt AES



3.2.3 – Padding Oracle to Decrypt AES

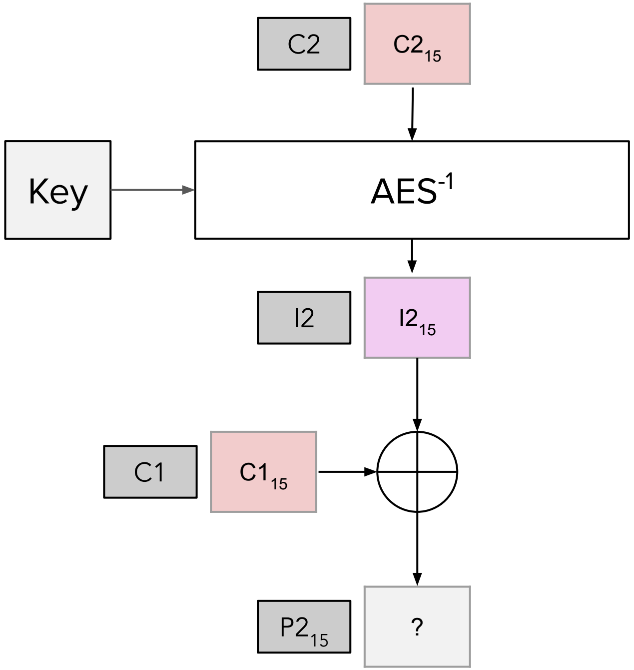


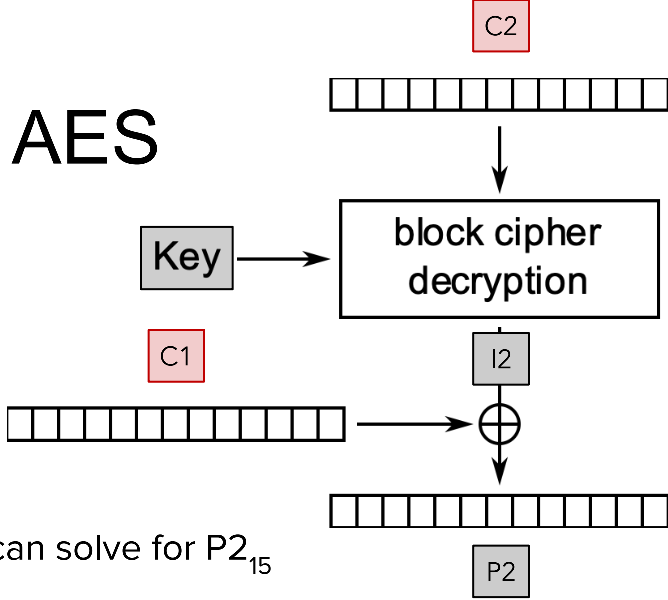


1. We keep trying different C1s until we get "valid padding, incorrect message". G is the specific $C1_{15}$ that give this message

2. Can solve for $I2_{15}$:
$I2_{15}$ ^ G = 0x10
$I2_{15}$ = 0x10 ^ G

3.2.3 – Padding Oracle to Decrypt AES





1. Now we know $I2_{15}$, we can solve for $P2_{15}$

2. We can solve for the plaintext:

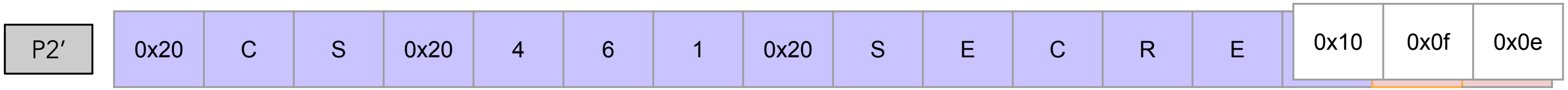
$I2_{15} \wedge C1_{15} = P2_{15}$

$0x10 \wedge G \wedge C1_{15} = P2_{15}$

We have solved for 1 character

3.2.3 – Padding Oracle to Decrypt AES

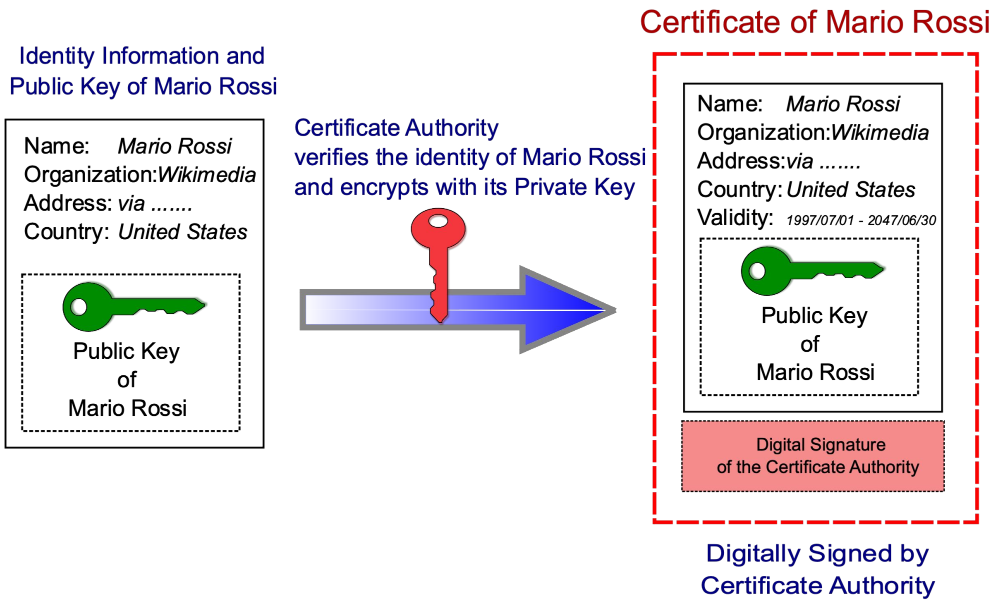
Fast forward a bit



Need to build this specific padding, which you can since you know $I2_{13\text{-}15}$

3.2.4 – Digital Certificates

- Certificate: certifies the ownership of a public key using an asymmetric signature. Protects against MITM!
- Certificate authority (CA): typically, the entity that issues the certificate.
- Subject: the entity the certificate is for. The subject can show this certificate to others to say. “Look, the Issuer says the public key in this certificate is really mine.”



3.2.4 – X.509 Certificates

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            75:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
        Signature Algorithm: md5WithRSAEncryption
        Issuer: CN=ece422
        Validity
            Not Before: Mar  1 00:00:00 2017 GMT
            Not After : Mar 27 00:00:00 2017 GMT
        Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2047 bit)
                Modulus:
                    52:3d:cc:11:9d:c8:e6:2d:0d:a5:38:f8:af:36:31:
                    47:40:95:a0:ae:9f:02:d6:5e:51:e3:ea:06:4d:bd:
                    […]:ba:c9:40:04:25
                Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
         84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
         eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
         […]:88:34:2e:94:33:
```

3.2.4 – X.509 Certificates

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            75:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
        Signature Algorithm: md5WithRSAEncryption
        Issuer: CN=ece422
        Validity
            Not Before: Mar  1 00:00:00 2017 GMT
            Not After : Mar 27 00:00:00 2017 GMT
        Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2047 bit)
                Modulus:
                    52:3d:cc:11:9d:c8:e6:2d:0d:a5:38:f8:af:36:31:
                    47:40:95:a0:ae:9f:02:d6:5e:51:e3:ea:06:4d:bd:
                    […]:ba:c9:40:04:25
                Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
        84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
        eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
        […]:88:34:2e:94:33:
```

Bytes that are signed by the CA
(using md5WithRSAEncryption)

Signature Produced by the CA
(using their private key)

3.2.4 – Colliding X.509 Certificates

Certificate 1:

```
Data:
    Version: 3 (0x2)
    Serial Number:
75:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
    Signature Algorithm: md5WithRSAEncryption
    Issuer: CN=ece422
    Validity
        Not Before: Mar  1 00:00:00 2017 GMT
        Not After : Mar 27 00:00:00 2017 GMT
    Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
            Public-Key: (2047 bit)
            Modulus:
                <<SOME MODULUS>>
            Exponent: 65537 (0x10001)
Signature Algorithm: md5WithRSAEncryption
     84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
     eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
     […]:88:34:2e:94:33:
```

Certificate 2:

```
Data:
    Version: 3 (0x2)
    Serial Number:
     75:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
    Signature Algorithm: md5WithRSAEncryption
    Issuer: CN=ece422
    Validity
        Not Before: Mar  1 00:00:00 2017 GMT
        Not After : Mar 27 00:00:00 2017 GMT
    Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
            Public-Key: (2047 bit)
            Modulus:
                <<DIFFERENT MODULUS>>
            Exponent: 65537 (0x10001)
Signature Algorithm: md5WithRSAEncryption
     84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
     eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
     […]:88:34:2e:94:33:
```

3.2.4 – Colliding X.509 Certificates

Certificate 1:

```
Data:
    Version: 3 (0x2)
    Serial Number:
75:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
    Signature Algorithm: md5WithRSAEncryption
    Issuer: CN=ece422
    Validity
        Not Before: Mar  1 00:00:00 2017 GMT
        Not After : Mar 27 00:00:00 2017 GMT
    Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
            Public-Key: (2047 bit)
            Modulus:
                <<SOME BYTES GENERATED BY FASTCOLL>>
            Exponent: 65537 (0x10001)
Signature Algorithm: md5WithRSAEncryption
     84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
     eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
     […]:88:34:2e:94:33:
```

Certificate 2:

```
Data:
    Version: 3 (0x2)
    Serial Number:
     175:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
    Signature Algorithm: md5WithRSAEncryption
    Issuer: CN=ece422
    Validity
        Not Before: Mar  1 00:00:00 2017 GMT
        Not After : Mar 27 00:00:00 2017 GMT
    Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
            Public-Key: (2047 bit)
            Modulus:
                <<DIFFERENT BYTES GENERATED BY FASTCOLL>>
            Exponent: 65537 (0x10001)
Signature Algorithm: md5WithRSAEncryption
     84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
     eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
     […]:88:34:2e:94:33:
```

Can we use fastcoll to generate two different moduli?

3.2.4 – Colliding X.509 Certificates

```
Certificate 1:
Data:
    Version: 3 (0x2)
    Serial Number:
75:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
    Signature Algorithm: md5WithRSAEncryption
    Issuer: CN=ece422
    Validity
        Not Before: Mar  1 00:00:00 2017 GMT
        Not After : Mar 27 00:00:00 2017 GMT
    Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
            Public-Key: (2047 bit)
            Modulus:
                <<SOME BYTES GENERATED BY FASTCOLL>>
            Exponent: 65537 (0x10001)
Signature Algorithm: md5WithRSAEncryption
     84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
     eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
     […]:88:34:2e:94:33:
```

```
Certificate 2:
Data:
    Version: 3 (0x2)
    Serial Number:
     175:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
    Signature Algorithm: md5WithRSAEncryption
    Issuer: CN=ece422
    Validity
        Not Before: Mar  1 00:00:00 2017 GMT
        Not After : Mar 27 00:00:00 2017 GMT
    Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
            Public-Key: (2047 bit)
            Modulus:
                <<DIFFERENT BYTES GENERATED BY FASTCOLL>>
            Exponent: 65537 (0x10001)
Signature Algorithm: md5WithRSAEncryption
     84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
     eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
     […]:88:34:2e:94:33:
```

Can we use fastcoll to generate two different moduli?
Yes, but how do we make sure the outputs of fastcoll are valid RSA moduli that we know p and q for?

3.2.4 – Colliding X.509 Certificates

```
Certificate 1:
    Data:
        Version: 3 (0x2)
        Serial Number:

75:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
        Signature Algorithm: md5WithRSAEncryption
        Issuer: CN=ece422
        Validity
            Not Before: Mar  1 00:00:00 2017 GMT
            Not After : Mar 27 00:00:00 2017 GMT
        Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2047 bit)
                Modulus:
                    <<SOME BYTES GENERATED BY FASTCOLL>>
                Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
         84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
         eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
         […]:88:34:2e:94:33:
```

```
Certificate 2:
    Data:
        Version: 3 (0x2)
        Serial Number:
         175:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
        Signature Algorithm: md5WithRSAEncryption
        Issuer: CN=ece422
        Validity
            Not Before: Mar  1 00:00:00 2017 GMT
            Not After : Mar 27 00:00:00 2017 GMT
        Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2047 bit)
                Modulus:
                    <<DIFFERENT BYTES GENERATED BY FASTCOLL>>
                Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
         84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
         eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
         […]:88:34:2e:94:33:
```

We'll just ask fastcoll to generate the first 1023 bits of the modulus, then we'll use Lenstra's attack to to generate a suffix that we can append to each of the 1023-bit collisions to make 2047-bit keys.

3.2.4 – Colliding X.509 Certificates

1. Create a valid prefix (green part)
2. Use fastcoll to generate MD5 collisions that share our prefix.
3. Use Lenstra's attack to to generate a suffix that we can append to each of the 1023-bit collisions to make 2047-bit keys.
4. Use the functions of mp3-certbuilder.py to generate two certificate with the public keys you found. They will have the same signature (verify with openssl).

prefix

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            75:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
        Signature Algorithm: md5WithRSAEncryption
        Issuer: CN=ece422
        Validity
            Not Before: Mar  1 00:00:00 2017 GMT
            Not After : Mar 27 00:00:00 2017 GMT
        Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2047 bit)
                Modulus:
                    52:3d:cc:11:9d:c8:e6:2d:0d:a5:38:f8:af:36:31:
                    47:40:95:a0:ae:9f:02:d6:5e:51:e3:ea:06:4d:bd:
                    […]:ba:c9:40:04:25
                Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
         84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
         eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
         […]:88:34:2e:94:33:
```

3.2.4 – Colliding X.509 Certificates

1. Create a valid prefix (green part)
   - The prefix should be generated starting from the tbs certificate bytes
   - Use the functions of `mp3-certbuilder.py` to generate a certificate with your net ID and a 2047-bit long modulus (pick `p` and `q` such that (p*q).bit_length() == 2047).
   - To get the tbs certificate bytes: `cert.tbs_certificate_bytes`.
   - You will need to modify the code of `mp3-certbuilder.py` so that the modulus starts at a 64-byte block (MD5 block size) in the certificate bytes.
   - Hint: alter the pseudonym field to get the right alignment.
   - Hint: use a hex editor to figure out where the modulus (`hex(p*q)`) starts and where to trim the tbs certificate bytes.
   - Trim the modulus and exponent parts from the tbs certificate bytes and keep the parts corresponding to the green box. The result is our prefix (its length must be a multiple of 64 bytes).
2. Use fastcoll to generate MD5 collisions that share our prefix.
   - Repeat a few times until both outputs (after the prefix), when read as a hex string, are 1023 bits long. Hint: assert that:
     `int(bytes_from_fastcoll.hex(), 16).bit_length() == 1023`
3. Use Lenstra's attack to to generate a suffix that we can append to each of the 1023-bit collisions to make 2047-bit keys.
   - This algorithm generates useful moduli, so you will learn `p` and `q`
4. Use the functions of mp3-certbuilder.py to generate two certificates with the public keys you found. They will have the same signature (verify with openssl).

prefix

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            75:64:4a:6d:1a:91:ec:d5:5b:39:29:35:50:1d:5b:63:0b:78:5e:d1
        Signature Algorithm: md5WithRSAEncryption
        Issuer: CN=ece422
        Validity
            Not Before: Mar  1 00:00:00 2017 GMT
            Not After : Mar 27 00:00:00 2017 GMT
        Subject: CN=rp8/pseudonym=unused, C=US, ST=Illinois
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2047 bit)
                Modulus:
                    52:3d:cc:11:9d:c8:e6:2d:0d:a5:38:f8:af:36:31:
                    47:40:95:a0:ae:9f:02:d6:5e:51:e3:ea:06:4d:bd:
                    […]:ba:c9:40:04:25
                Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
         84:a9:ca:c3:42:8e:3d:f0:f0:90:f4:1a:79:a0:d8:48:ec:3e:
         eb:df:58:d2:09:e4:da:a1:4b:55:09:0f:dc:1e:75:79:69:f1:
         […]:88:34:2e:94:33:
```

3.2.4 – Lenstra's Attack

The symbol | means
"divides evenly"

- Let b_1 and b_2 be the outputs of fastcoll.
- Generate random primes p_1 and p_2 of approximately 512 bits [we suggest shorter, e.g., 400], such that e is coprime to $p_1 - 1$ and $p_2 - 1$;
  - [Hint: use Crypto.Util.number.getPrime(400)]
- Compute b_0 between 0 and p_1p_2 such that $p_1 \mid b_12^{1024} + b_0$ and $p_2 \mid b_22^{1024} + b_0$ (by the Chinese Remainder Theorem);
  - [Hint: b_0 = getCRT(b_12^{1024}, b_22^{1024}, p_1, p_2)]
- Let k run through 0,1,2, ..., and for each k compute $b = b_0 + kp_1p_2$; check whether both $q_1 = (b_12^{1024} + b)/p_1$ and $q_2 = (b_22^{1024} + b)/p_2$ are primes, and whether e is coprime to both $q_1 - 1$ and $q_2 - 1$;
  - [Hint: use Crypto.Util.number.isPrime]
- When k has become so large that $b \geq 2^{1024}$, restart with new primes p_1, p_2;
- When primes q_1 and q_2 have been found, stop, and output $n_1 = b_12^{1024} + b$ and $n_2 = b_22^{1024} + b$ (as well as p_1, p_2, q_1, q_2).

3.2.4 – Chinese Remainder Theorem

```
# b1_exp = b1*2^1024
# b2_exp = b2*2^1024
def getCRT(b1_exp, b2_exp, p1, p2):
    N = p1 * p2
    invOne = number.inverse(p2, p1)
    invTwo = number.inverse(p1, p2)
    return -(b1_exp * invOne * p2 + b2_exp * invTwo * p1) % N
```

3.3.1 – RSA Decryption

- e - public prime
- n - public modulus
- d - secret
- m - plaintext
- c - ciphertext

- Encryption: $c = m^e \bmod(n)$
- Decryption: $m = c^d \bmod(n)$

3.3.1 – RSA Decryption

- e - public prime
- n - public modulus
- d - secret
- m - plaintext
- c - ciphertext

- Encryption: $c = m^e \bmod(n)$
- Decryption: $m = c^d \bmod(n)$

- $n = p * q$
- p and q are *random* large primes
- $d = e^{-1} \pmod{(p-1)(q-1)}$ (modular multiplicative inverse) [Crypto.Util.number.inverse]
- If you can factor n (i.e., recover p and q), you can get the secret key d

3.3.1 – RSA Decryption

- $n = p * q$
- p and q are large primes
- $d = e^{-1} \pmod{(p-1)(q-1)}$ (modular multiplicative inverse) [Crypto.Util.number.inverse]
- If you can factor n (i.e., recover p and q), you can get the secret key d

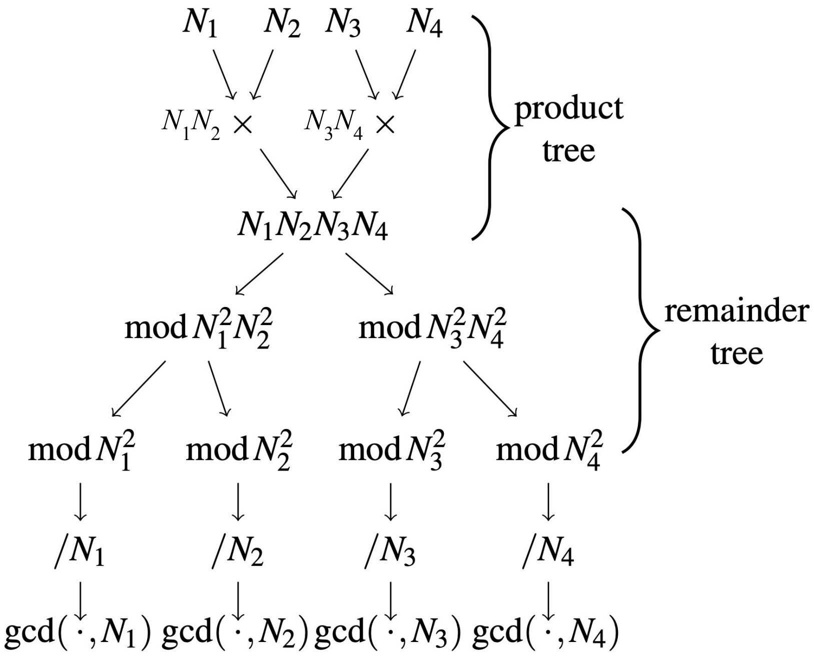
- <u>Factoring n is hard</u>
- However, finding the Greatest Common Divisor (GCD) of two numbers can be done quickly!
- What if two users happened to choose the same values for p *or* q when they generated their moduli? Then it is trivial to compute their GCD and factor both moduli!

3.3.1 – Mining Ps and Qs

- $n = p * q$
- p and q are large primes
- $d = e^{-1} \pmod{(p-1)(q-1)}$ (modular multiplicative inverse) [Crypto.Util.number.inverse]
- If you can factor n (i.e., recover p and q), you can get the secret key d

- We've given you 10,000 moduli (n)
- One of the moduli is from the key that can decrypt your ciphertext.
- You need to find which moduli share a p or q with each other by using the Euclidean GCD.
- A pairwise comparison is $O(N^2)$ comparisons (100 million).
- You need to implement a product / remainder tree like Heninger et al.

3.3.1 – Mining Ps and Qs



- We've given you 10,000 moduli (n)
- One of the moduli is from the key that can decrypt your ciphertext.
- You need to find which moduli share a p or q with each other by using the Euclidean GCD.
- A pairwise comparison is $O(N^2)$ comparisons (100 million).
- You need to implement a product / remainder tree like Heninger et al.

3.3.1 – Mining Ps and Qs

- Your code will find the p and q values of multiple moduli, allowing you to recover multiple secret keys d. For each of them try:

```
# Construct the RSA key for the computed values
key = RSA.construct((int(modulus), int(65537), int(d)))

try:
    # Try to decrypt the ciphertext with the built key
    plaintext = pbp.decrypt(key, ciphertext)
    print(plaintext)

except ValueError:
    # The PBP decrypt function will throw a value error for an incorrect RSA key
    pass
```

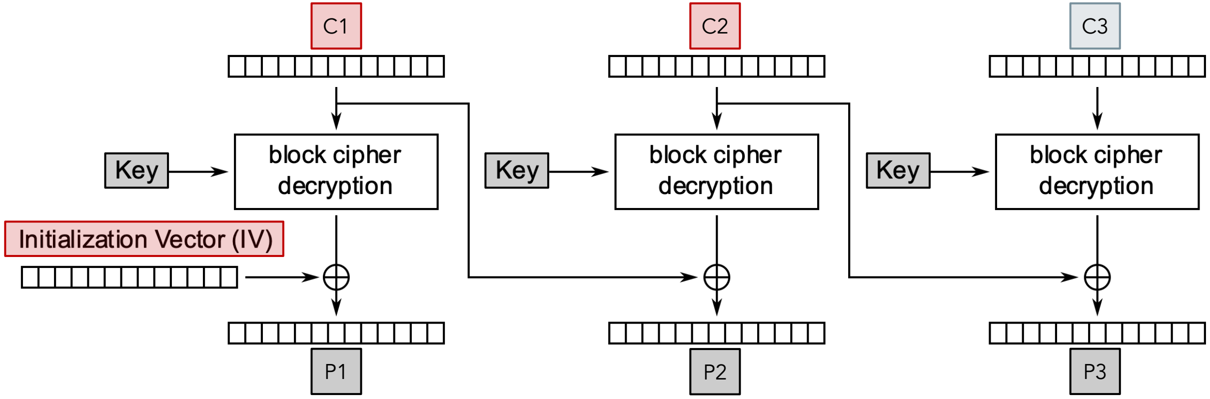
Summary

- 3.2.2 – generating two files with the same MD5 hash (collisions)
- 3.2.3 – decrypt an AES ciphertext w/o the key (padding oracle)
- 3.2.4 – create a pair of distinct (but valid) certificates, which both share the same signature.
- 3.3.1 – decrypt RSA ciphertext by factoring weak moduli.

Additional Slides

- These slides are not presented in discussion but they might help if you started early on the exercises of CP2.

3.2.3 – Padding Oracle to Decrypt AES

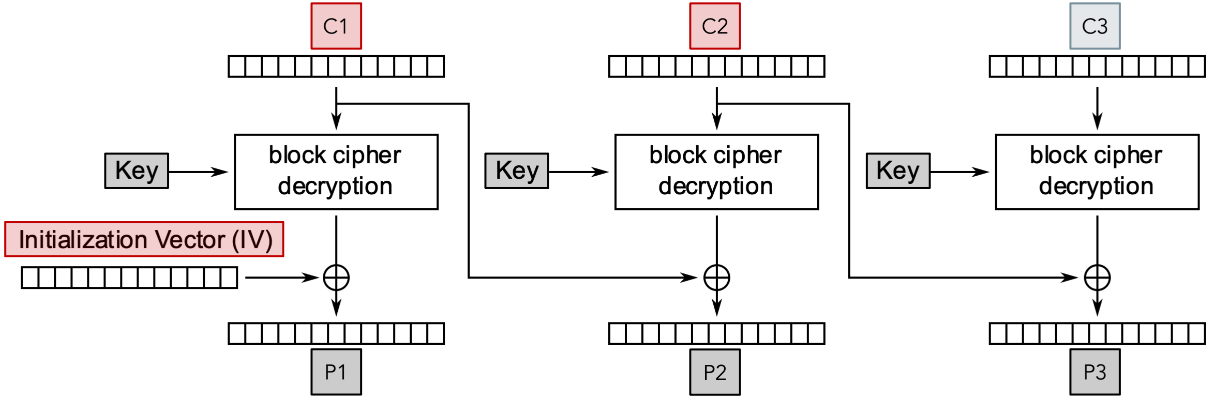


$P3 = Dec(C3, k) \oplus C2$

$P3[15] = Dec(C3, k)[15] \oplus C2[15]$

1. If the plaintext (P3) does not end with valid padding (\x10\x0f…), error 500
2. If the plaintext doesn't match the expected string, error 404
3. If the plaintext matches what the server was expecting, success

3.2.3 – Padding Oracle to Decrypt AES



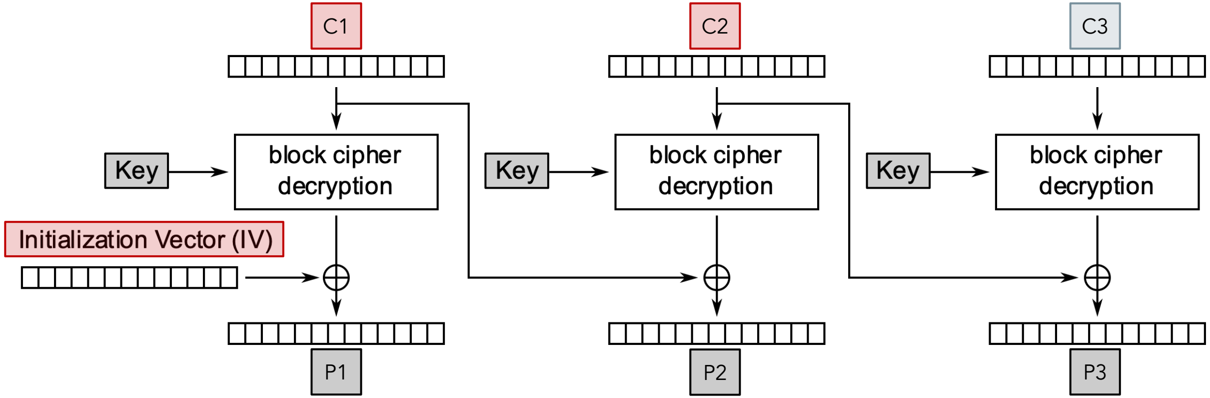
P3 = Dec(C3, k) ⊕ C2

P3[15] = Dec(C3, k)[15] ⊕ C2[15]

If P3[15] == '\x10':
    padding is valid (error 404)
else:
    padding is invalid (error 500)

1. If the plaintext (P3) does not end with valid padding (\x10\x0f…), error 500
2. If the plaintext doesn't match the expected string, error 404
3. If the plaintext matches what the server was expecting, success

3.2.3 – Padding Oracle to Decrypt AES



1. If the plaintext (P3) does not end with valid padding (\x10\x0f...), error 500
2. If the plaintext doesn't match the expected string, error 404
3. If the plaintext matches what the server was expecting, success

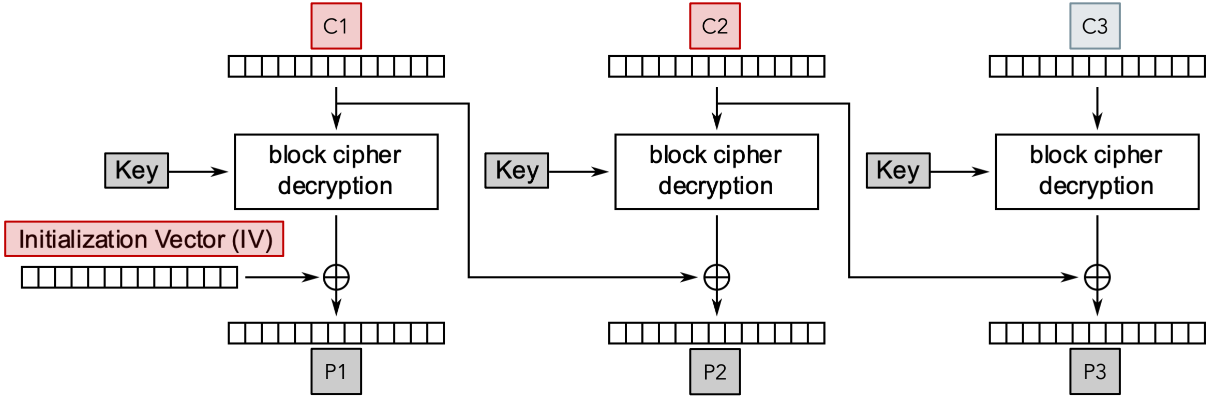
P3 = Dec(C3, k) ⊕ C2

P3[15] = Dec(C3, k)[15] ⊕ C2[15]

If P3[15] == '\x10':
    padding is valid (error 404)
else:
    padding is invalid (error 500)

There is only one value of C2[15] which results in P3[15] == '\x10'. We can try all possible values of C2[15] until we get 404.

3.2.3 – Padding Oracle to Decrypt AES



P3 = Dec(C3, k) ⊕ C2

P3[15] = Dec(C3, k)[15] ⊕ C2[15]

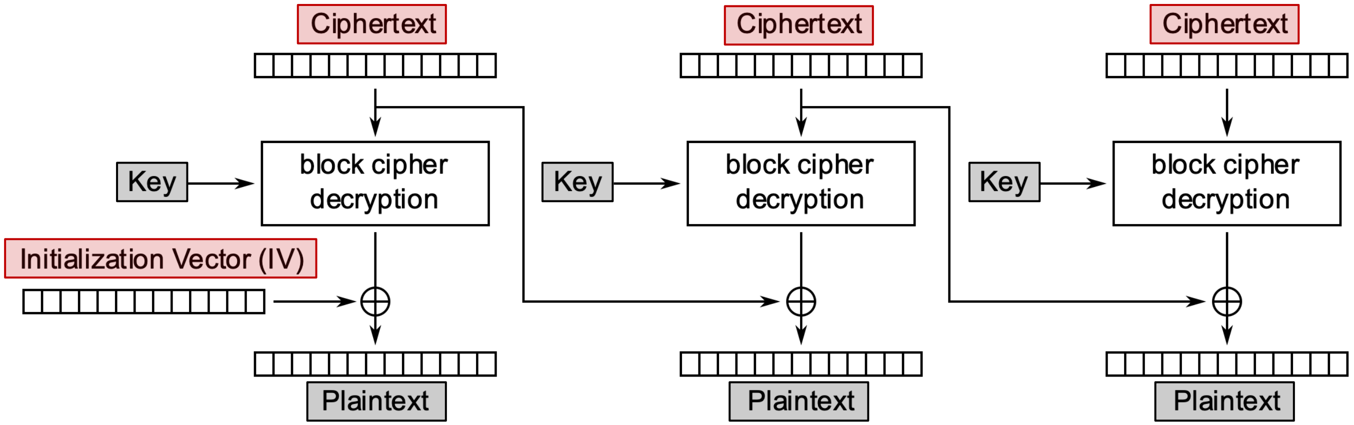
If P3[15] == '\x10':
    padding is valid (error 404)
else:
    padding is invalid (error 500)

There is only one value of C2[15] which results in P3[15] == '\x10'. We can try all possible values of C2[15] until we get 404.

1. If the plaintext (P3) does not end with valid padding (\x10\x0f...), error 500
2. If the plaintext doesn't match the expected string, error 404
3. If the plaintext matches what the server was expecting, success

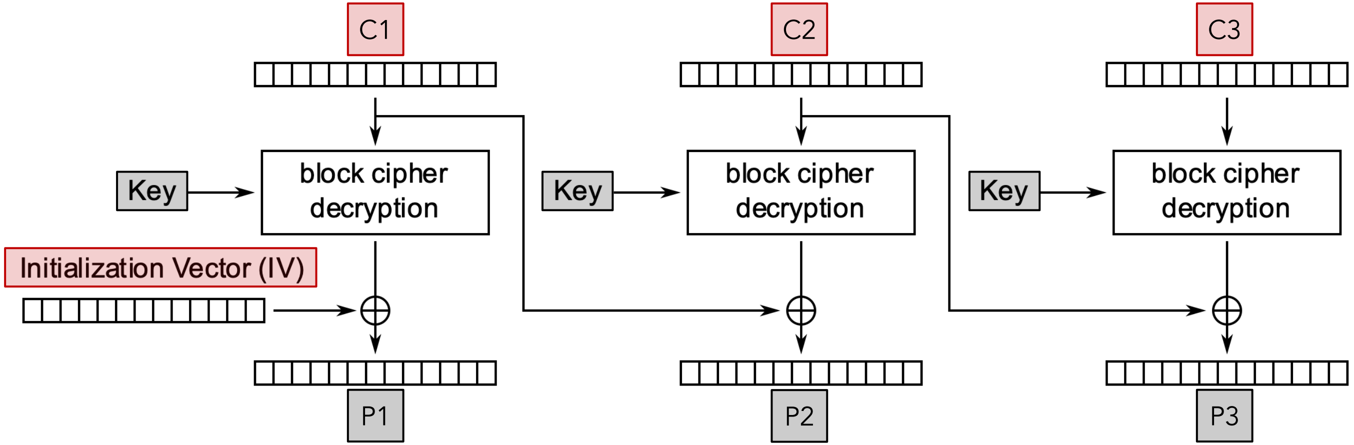
Dec(C3, k)[15] = P3[15] ⊕ C2[15]

3.2.3 – Padding Oracle to Decrypt AES



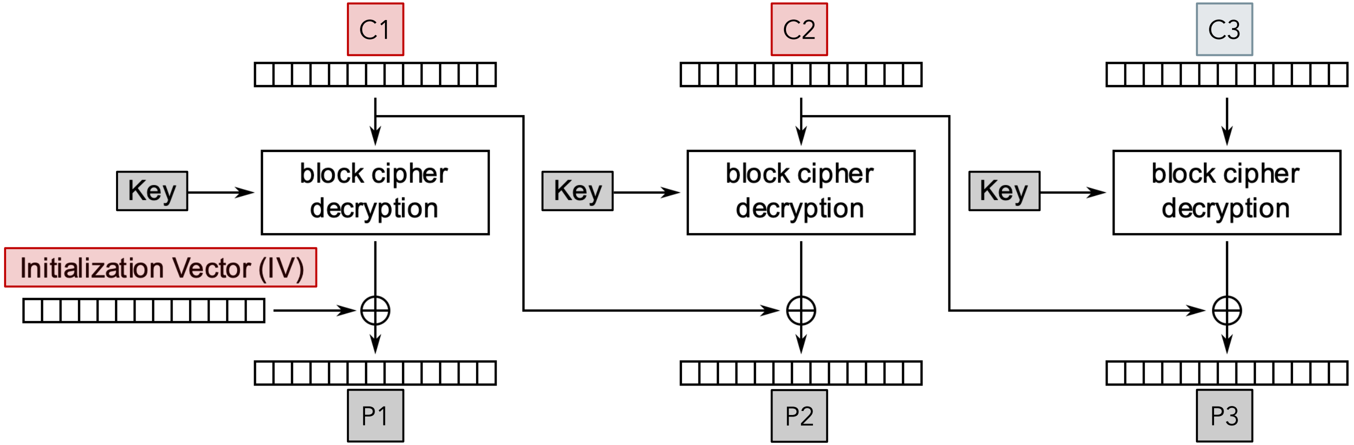
1. If the plaintext does not end with valid padding (\x10\x0f…), error 500
2. If the plaintext doesn't match the expected string, error 404
3. If the plaintext matches what the server was expecting, success

3.2.3 – Padding Oracle to Decrypt AES



1. If the plaintext (P3) does not end with valid padding (\x10\x0f...), error 500
2. If the plaintext doesn't match the expected string, error 404
3. If the plaintext matches what the server was expecting, success

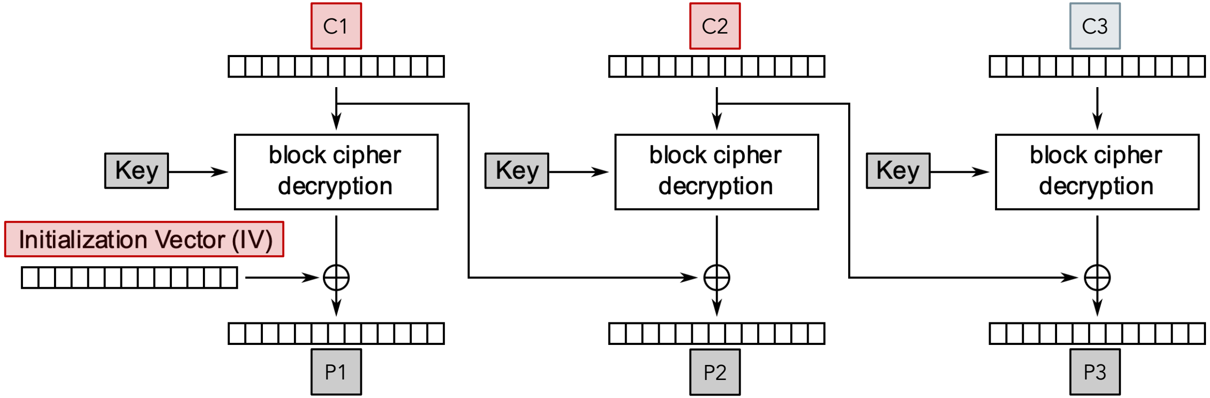
3.2.3 – Padding Oracle to Decrypt AES



<u>Example</u>: let C3 be the first block of the ciphertext we gave you, while C1 and C2 are arbitrary.

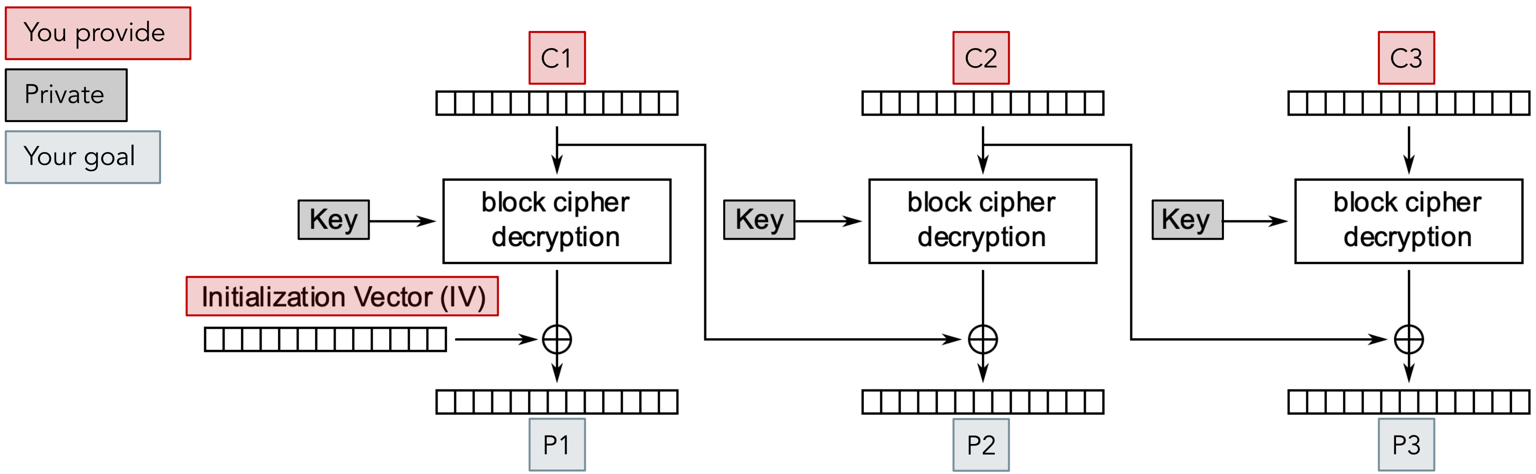
1. If the plaintext (P3) does not end with valid padding (\x10\x0f...), error 500
2. If the plaintext doesn't match the expected string, error 404
3. If the plaintext matches what the server was expecting, success

3.2.3 – Padding Oracle to Decrypt AES



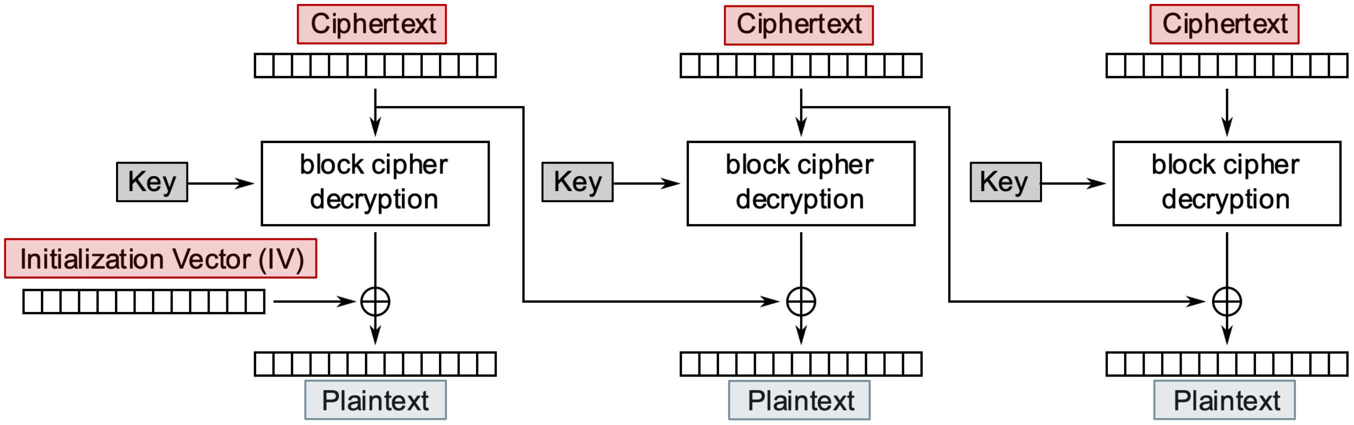
1. If the plaintext (P3) does not end with valid padding (\x10\x0f...), error 500
2. If the plaintext doesn't match the expected string, error 404
3. If the plaintext matches what the server was expecting, success

3.2.3 – Padding Oracle to Decrypt AES



$$P3 = Dec(C3, k) \oplus C2$$

3.2.3 – Padding Oracle to Decrypt AES



3.2.3 – Padding Oracle to Decrypt AES