Ataque de Hastad

April 30, 2022

1 Hastad Broadcast Attack

- Alef Pinto Keuffer, a91683
- Beatriz Fernandes Oliveira, a91640
- Catarina Martins Sá Quintas, a91650

2 RSA Background

Using code from [1]:

[1]:
```
# from W. A. Stein, Elementary number theory: primes, congruences, and secrets␣
↪a computational approach. New York, NY: Springer, 2009. p. 58.

def rsa(bits):
    # only prove correctness up to 1024 bits
    proof = (bits <= 1024)
    p = next_prime(ZZ.random_element(2**(bits//2+1)),
            proof=proof)
    q = next_prime(ZZ.random_element(2**(bits//2+1)),
            proof=proof)
    n = p*q
    phi_n = (p-1)*(q-1)
    while True:
        e = ZZ.random_element(1,phi_n)
        if gcd(e,phi_n) == 1: break
    d = lift(Mod(e,phi_n)^(-1))
    return d, n, e

def encrypt(m, n, e):
    assert m < n # message must be in  /n
    return lift(Mod(m,n)^e)

def decrypt(c, d, n):
    return lift(Mod(c,n)^d)
```

[2]:
```
def rsa_e(bits,e,used_primes=set()):
    # adapted from @rsa and https://www.di-mgt.com.au/rsa_alg.html
```

```
    # only prove correctness up to 1024 bits
    assert e % 2
    proof = (bits <= 1024)
    while True:
        p = next_prime(ZZ.random_element(2**(bits//2+1)),
                proof=proof)
        if p in used_primes:
            continue
        if p % e != 1:
            break
    while True:
        q = next_prime(ZZ.random_element(2**(bits//2+1)),
                proof=proof)
        if q in used_primes:
            continue
        if q % e != 1:
            break
    used_primes.update({p,q})
    n = p*q
    phi_n = (p-1)*(q-1)
    d = lift(Mod(e,phi_n)^(-1))
    return d, n, e
```

2.1 Encoding

In order to use the RSA cryptosystem to encrypt messages, it is necessary to encode them as a sequence of numbers of size less than n = pq. [1]

Based on code from [3]:

[3]:
```
# based on stackoverflow.com/questions/55407713/
 ↪how-to-encode-a-text-string-into-a-number-in-python
def encode(s):
    s = str(s)
    mybytes = b'\x01' + s.encode('utf8')  # Pad with 1 to preserve trailing␣
 ↪zeroes
    return int.from_bytes(mybytes, 'big')

def decode(n):
    n = int(n)
    recoveredbytes = n.to_bytes((n.bit_length() + 7) // 8, 'big')
    return recoveredbytes[1:].decode('utf8') # Strip pad before decoding
```

3 Håstad's broadcast attack

Suppose Eve intercepts C_1, C_2, and C_3, where $C_i \equiv M^3 \pmod{N_i}$. We may assume $\gcd(N_i, N_j) = 1$ for all i,j (otherwise, it is possible to compute a factor of one of the numbers N_i by computing $\gcd(N_i, N_j)$. By the Chinese remainder theorem, she may compute $C \in \mathbb{Z}^*_{N_1 N_2 N_3}$ such that $C \equiv C_i \pmod{N_i}$. Then $C \equiv M^3 \pmod{N_1 N_2 N_3}$; however, since $M < N_i$ for all i, we have $M^3 < N_1 N_2 N_3$. Thus $C = M^3$ holds over the integers, and Eve can compute the cube root of C to obtain M. [2]

[4]:
```
def hastad(C_list,N_list):
    # assumes len(C_list) is the exponent used to encrypt the message.
    assert len(C_list) == len(N_list)
    return CRT_list(C_list,N_list).nth_root(len(C_list))
```

3.1 Examples

[5]:
```
bits = 512
```

[6]:
```
class Test:
    def __init__(self,bits=4096):
        self.__keys = {}
        self.bits = 4096

    def keys(self,e):
        if e not in self.__keys:
            self.__keys[e] = [rsa_e(self.bits,e) for _ in range(e)]
        return self.__keys[e]

    def hastad(self,message,e=3):
        emessage = encode(message)
        cyphertexts = [encrypt(emessage,k[1],k[2]) for k in self.keys(e)]
        dmessage = decode(hastad(cyphertexts,[k[1] for k in self.keys(e)]))
        print(dmessage)

    def encipher_decipher(self,message):
        d,N,e = rsa(self.bits)
        cyphertext = encipher(message,N,e)
        print(m_decipher(cyphertext,d,N))

    def m_hastad(self,message,e=3):
        cyphertexts = [encipher(message,k[1],k[2]) for k in self.keys(e)]
        attacked = m_hastad(cyphertexts,[k[1] for k in self.keys(e)])
        print(decipher(attacked))

def break_encoded(e_int,max_block_size):
    assert max_block_size > 8
    e = str(e_int)
    len_e = len(e)
    r = [0]
    i = 0
    while i < len_e:
```

```python
        j = i + max_block_size
        while j < len_e and e[j] == '0':
            j -= 1
        r.append(j)
        i = j
    res = [e[r[i]:r[i+1]] for i in range(0,len(r)-1)]
    assert all([s[0] != '0' for s in res])
    assert ''.join(res) == e
    return [int(i) for i in res]

def encipher(M,N,e,encoding=True,min_N=None):
    if encoding == True:
        enc = encode(M)
        max_block_size = int(round(len(str(N)),-2))
        return [encrypt(m,N,e) for m in break_encoded(enc,max_block_size)]
    else:
        return [encrypt(m,N,e) for m in M]

def decipher(cyphertext,d=None,N=None):
    r = cyphertext
    if d is not None and N is not None:
        r = [decrypt(c,d,N) for c in cyphertext]
    return decode(''.join([str(c) for c in r]))

def m_hastad(C_list_list,N_list):
    assert len(C_list_list) == len(N_list)
    assert len(set(len(cs) for cs in C_list_list)) == 1
    r = []
    for i in range(len(C_list_list[0])):
        r.append(hastad([cs[i] for cs in C_list_list],N_list))
    return r
```

[7]:
```python
T = Test(512)
```

3.1.1 Examples for $e = 3$

Example 1

[8]:
```python
T.hastad('alef',3)
```

```
alef
```

Example 2

[9]:
```python
T.hastad('3rd example.',3)
```

```
3rd example.
```

Example 3

[0]:
```
T.hastad('m6KKkC7QTL263VX6',3)
```

3.1.2 Examples for $e = 5$

Example 4

[10]:
```
T.hastad('alef',5)
```

```
alef
```

3.1.3 Examples for $e = 17$

Example 5

[11]:
```
message = '''The RSA algorithm is named after Ron Rivest, Adi Shamir and Len␣
 ↪Adleman, who invented it in 1977.
RSA use a number of concepts from cryptography:

    - A one-way function that is easy to compute; finding a function that␣
 ↪reverses it, or computing this function is very difficult.
    - RSA uses a concept called discrete logarithm. This works much like the␣
 ↪normal logarithm: The difference is that only whole numbers are used, and in␣
 ↪general, a modulus operation is involved. As an example ax=b, modulo n. The␣
 ↪discrete logarithm is about finding the smallest x that satisfies the␣
 ↪equation, when a b and n are provided.'''

T.m_hastad(message,17)
```

```
The RSA algorithm is named after Ron Rivest, Adi Shamir and Len Adleman, who
invented it in 1977.
RSA use a number of concepts from cryptography:

    - A one-way function that is easy to compute; finding a function that
reverses it, or computing this function is very difficult.
    - RSA uses a concept called discrete logarithm. This works much like the
normal logarithm: The difference is that only whole numbers are used, and in
general, a modulus operation is involved. As an example ax=b, modulo n. The
discrete logarithm is about finding the smallest x that satisfies the equation,
when a b and n are provided.
```

[12]:
```
message = '''The Rivest, Shamir, Adleman (RSA) cryptosystem is an example of a␣
 ↪public key cryptosystem. RSA uses a public key to encrypt messages and␣
 ↪decryption is performed using a corresponding private key. We can distribute␣
 ↪our public keys, but for security reasons we should keep our private keys to␣
 ↪ourselves. The encryption and decryption processes draw upon techniques from␣
 ↪elementary number theory. The algorithm below is adapted from page 165 of␣
 ↪[TrappeWashington2006]. It outlines the RSA procedure for encryption and␣
 ↪decryption.
```

```
Choose two primes p and q and let n=pq.

Let e Z be positive such that gcd(e, (n))=1.

Compute a value for d Z such that de 1(mod (n)).

Our public key is the pair (n,e) and our private key is the triple (p,q,d).

For any non-zero integer m<n, encrypt m using c me(modn).

Decrypt c using m cd(modn).

The next two sections will step through the RSA algorithm, using Sage to␣
↪generate public and private keys, and perform encryption and decryption␣
↪based on those keys.'''

T.m_hastad(message,17)
```

```
The Rivest, Shamir, Adleman (RSA) cryptosystem is an example of a public key
cryptosystem. RSA uses a public key to encrypt messages and decryption is
performed using a corresponding private key. We can distribute our public keys,
but for security reasons we should keep our private keys to ourselves. The
encryption and decryption processes draw upon techniques from elementary number
theory. The algorithm below is adapted from page 165 of [TrappeWashington2006].
It outlines the RSA procedure for encryption and decryption.

Choose two primes p and q and let n=pq.

Let e Z be positive such that gcd(e, (n))=1.

Compute a value for d Z such that de 1(mod (n)).

Our public key is the pair (n,e) and our private key is the triple (p,q,d).

For any non-zero integer m<n, encrypt m using c me(modn).

Decrypt c using m cd(modn).

The next two sections will step through the RSA algorithm, using Sage to
generate public and private keys, and perform encryption and decryption based on
those keys.
```

3.1.4 e interceptions are required only in the worst case

It is possible that $M^e < \prod S$, where $S \subset \{N_0, \dots, N_e\}$

[13]:
```
e = 5
keys_ = [rsa_e(bits,e) for _ in range(e)]
```

[14]:
```
keys = keys_[:3]
def hastad(C_list,N_list,e=None):
    # assumes len(C_list) is the exponent used to encrypt the message.
    if e is None:
        e = len(C_list)
    assert len(C_list) == len(N_list)
    return CRT_list(C_list,N_list).nth_root(e)

emessage = encode('RSA can be can be vulnerable.')
cyphertexts = [encrypt(emessage,k[1],k[2]) for k in keys]
dmessage = decode(hastad(cyphertexts,[k[1] for k in keys],e))
print(dmessage)
```

```
RSA can be can be vulnerable.
```

4 References

[1] W. A. Stein, Elementary number theory: primes, congruences, and secrets a computational approach. New York, NY: Springer, 2009. p. 58.

[2] G. Durfee, "CRYPTANALYSIS OF RSA USING ALGEBRAIC AND LATTICE METHODS," p. 25.

[3] stackoverflow.com/questions/55407713/how-to-encode-a-text-string-into-a-number-in-python.