Padding Schemes for RSA and their Security

Calvin Ryan D'Souza
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
cd5184@cs.rit.edu

***Abstract*—One of the oldest algorithms used for data transmission and digital signatures is the Rivest-Shamir-Adleman (RSA) public-key cryptosystem. The security of RSA banks on difficulty of factoring the product of two large prime numbers i.e., the factorization problem. The paper experiments with generation of safe and basic primes for RSA implementation. RSA is computationally expensive and commonly only used for key transmission/ digital signatures. The paper explores the feasibility of using RSA as a data transmission encryption scheme. It is a deterministic cryptosystem, which makes it susceptible chosen-plaintext attacks. To mitigate this, RSA is almost always employed along with padding schemes. Padding schemes introduce a random component into RSA, thus making the algorithm probabilistic. The paper explores security of padding schemes used with RSA.**

I. Introduction

Digital signatures are widely used on the internet for verifying the identity of websites. The Rivest-Shamir-Adleman encryption scheme [1] commonly known as RSA is one of the main algorithms used for implementing digital signatures. RSA is based on the factorization problem. The security of RSA is based on the fact that the product of 2 large prime numbers is extremely difficult to factor. RSA is computationally expensive since it is implemented with 2048/4096-bit numbers. The primitive versions of RSA as mentioned in [2] are usually referred to as textbook implementations.

Textbook RSA is a deterministic cryptosystem (i.e., has no random component) which makes it susceptible to chosen-plaintext attacks. This means it is not semantically secure. Thus, in real-world scenarios, RSA is almost always implemented with a padding scheme. Padding Schemes add a random component to RSA and makes it probabilistic in nature. Before actually encrypting the message, the message is padded with some random string. This forms an embedded string which is then sent for RSA encryption. Commonly used padding schemes with RSA are PKCS #1 v1.5 (RSAES-PKCS1-v1_5) and Optimal Asymmetric Encryption Padding (RSAES-OAEP) [2]. The paper implements these schemes in accordance with [3] for experimental purposes.

Prime numbers generated are categorized as basic primes and safe primes for each RSAES padding scheme. These are generated using random number/bit generators and primality tests as mentioned in [3]. Safe primes are generated since prime numbers can often be factorized using Pollard's p-1 algorithm [4]. This does not usually occur in case of prime numbers generated for cryptographic purposes. But if it does occur, it will be detrimental to the RSAES implementations even though a padding scheme is employed. The paper provides insight into implementing RSA with safe primes versus basic primes and the computational complexities related to them.

RSAES-PKCS1-v1_5 has been broken as demonstrated in [5]. However, it is still recommended as usable by FIPS 186-4 [3] and FIPS 186-5 [6]. The paper executes and analyses this attack and highlights that the security of RSAES-PKCS1-v1_5 can be easily compromised. The paper also provides analysis on how RSAES-OAEP reduces message length for RSA, thus increasing computational costs since a higher bit RSA implementation is required. SHAKE-128/246 is recommended as a Mask generation function in [6]. The paper proposes use of SHAKE-128/256 as a viable hash function for RSAES-OAEP.

II. Primes and Textbook RSA

A. Safe Primes Vs Basic Primes

The RSA modulus used for encryption and decryption is the product of two randomly generated large prime numbers. RSA security is based on the premise that this modulus is hard to factorize. However, for a prime number p if p-1 has many small factors, it is possible to find p given p-1. This is called Pollard's Algorithm. This introduces the concept of safe primes.

For a prime p, if p-1 can be expressed as 2 * prime (m), then p is called a Safe prime. The prime number m is known as a Sophie Germain prime. Some examples of Sophie Germain primes are 2, 3, 11, 23 and their respective Safe primes are 5, 7, 23, 47. Prime numbers for RSA are generated using cryptographically safe random number generators. Thus, the primes numbers used for generating the RSA modulus are often not susceptible to Pollards Algorithm. However, if the prime numbers are chosen with care, it could be detrimental to RSA.

As part of this project, we have compared the cost of implementing RSA with basic primes and safe primes. The prime number generation algorithm used as part of the project generates a set of primes and then filters them for safe primes.

B. Key Generation Scheme for RSA and Textbook RSA

As mentioned earlier, the basic RSA encryption scheme without padding is known as Textbook RSA. Textbook n-bit RSA functions as follows:-

1) Randomly generate two n/2-bit prime numbers (*p* and *q*). Compute $n = p*q$.
2) *n* is the modulus for RSA.
3) Compute Φ (n) = (*p*-1)(*q*-1).
4) Choose public-key exponent *e*. *e* is chosen such that GCD(*e*, Φ (*n*)) = 1.
5) Compute private-key exponent, $d \equiv e^{-1} \; mod \; \Phi(n)$.
6) The public-key pair for RSA = (*e*, *n*) and private-key pair = (*d*, *n*).

Consider plaintext (p) and corresponding ciphertext (c). Since we have the key-pairs, encryption and decryption in textbook RSA is as follows:-

- Encryption: $c = m^e \; mod \; n$
- Decryption: $m = c^d \; mod \; n$

C. Padding Schemes

RSA in the real-world is used with Padding Schemes given its deterministic nature. FIPS 186-4 [3] which outlines Digital Signature Standards (DSS) recommends PKCS #1 v1.5 and OAEP as viable Padding schemes with RSA. Padding schemes modify the input message to generate encoded messages which are then used for encryption by RSA. A basic schematic for encryption and decryption using a Padding scheme is shown in Fig. 1.

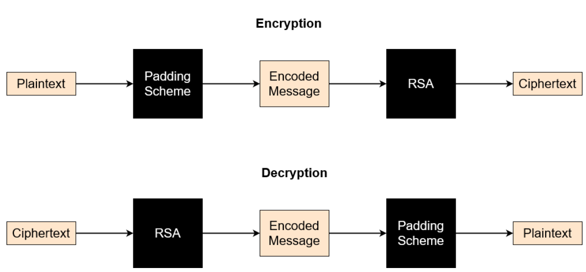

Fig. 1. Padding Scheme Architecture

Thus, on the sender's end, the input message is encoded by the padding scheme and then encrypted using RSA. On the receiving end, the ciphertext is decrypted using RSA and then decoded using the padding scheme.

III. Public-Key Cryptography Standard (PKCS) #1 v1.5

Public-Key Cryptography Standard (PKCS) #1 v1.5 is one of the Padding Schemes which is employed with RSA. It was officially published in RFC2313 [7] in March 1998. It is a simple padding scheme but susceptible to timing attacks. PKCS #1 v1.5 is considered broken, since there is a known attack against it called Bleichenbacher's [5] attack. Despite being broken, FIPS 186-4 [3] still recommends PKCS #1 v1.5 as a viable Padding Scheme with RSA. As explained above, padding schemes are used for encoding the input message before it is sent for encryption.

The input string for RSA is usually the same byte length as the RSA modulus. Let this byte length be k. PKCS #1 v1.5 has a minimum padding length of 11 bytes. Out of these 11 bytes, the starting 2 bytes (0x00 and 0x02) and last byte (0x00) of the padding are fixed. Hence, the randomized padding is at least 8 bytes long. This results in the message length being restricted to a maximum of k-11 bytes. Encoding using PKCS #1 v1.5 has the following steps (Fig. 2):-

1) Let the number of bytes in the RSA modulus be *k*.
2) Length checking the input message (*m*): If $len(m) > k$ - 11 bytes, report error and stop.
3) Create encoded message as follows:-
   a) Add *0x00* as 1st byte
   b) Append *0x02* as 2nd byte
   c) Append (*k* - 8 - *len*(*m*)) non-zero bytes as randomized padding bytes.
   d) Append *0x00* as separation byte
   e) Append input message *m* to form final encoded message.

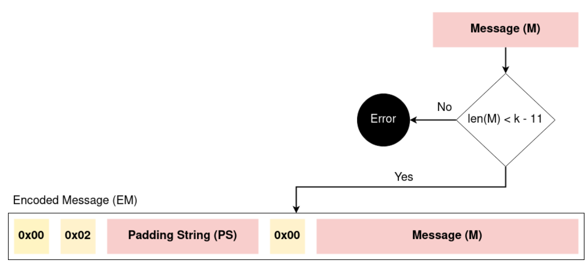

Fig. 2. Encoding plaintext with PKCS #1 v1.5

The encoded message is then encrypted using public-key pair and RSA to generate the ciphertext. On the receiver's end, the ciphertext is decrypted using private-key pair and RSA to generate a possible encoded message. As mentioned earlier, PKCS # 1 v1.5 is susceptible to attacks and thus the encoded message needs to be validated for correctness. After the encoded message is successfully validated, the original plaintext is recovered. The encoded message is decoded and validated as follows (Fig. 3):-

1) Length checking encoded message (*EM*): If *len(EM)* != *k* or $len(EM) < 11$, report error and stop.
2) If 1st byte is not *0x00*, report error and stop. Else, discard byte.
3) If 1st byte is not *0x02*, report error and stop. Else, discard byte.
4) Traverse bytes until *0x00* is encountered. If number of bytes traversed < 8, report error and stop. Else, discard bytes.
5) If 1st byte is not *0x00*, report error and stop. Else, discard byte.
6) Since all padding bytes are discarded, original input message can now be extracted.

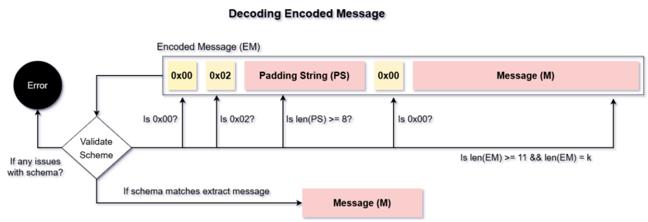

Fig. 3. Encoding EM with PKCS #1 v1.5

A. Bliechenbacher's Attack

Bliechenbacher [5] attack exploits the fact that PKCS starts with 0x00 0x02. The attack is works when the attacker has access to an oracle which returns if a given ciphertext conforms with PKCS #1 v1.5 [7]. The attack is highly potent and renders PKCS #1 v1.5 [7] is broken. However, PKCS #1 v1.5 is still considered as a viable Padding Scheme by [3] and [6]. For a given ciphertext (c), the attack functions as follows [5]:-

1) Find s, such that s^e.c is PKCS compliant.
2) Now, we know that:-

$$0x00||0x02||B \leq s^e.c < 0x00||0x03||B$$

3) For RSA with k-byte modulus:-

$$B = 2^{8(k-2)}$$

4) Since c = m^e mod n, we have:

$$0x00||0x02||B \leq (ms)^e modn < 0x00||0x03||B$$

5) Searching for PKCS Conforming Messages:-
   a) Starting with i = 1, we calculate s_i = *n/3B*, such that:-

   $$s_i{}^e.c \mod n \textit{ is PKCS compliant}$$

   b) Suppose multiple intervals are found, search for smallest integer $s_i < s_{i-1}$, such that s_i forms a PKCS compliant ciphertext.
   c) If there is only one interval left (i.e., $M_{i-1} = [a, b]$), then choose small integer values $r_i s_i$ such that:

   $$r_i = \frac{bs_{i-1} - 2B}{n}$$

   and

   $$\frac{2B + r_i n}{b} \leq s_i < \frac{3B + r_i n}{a}$$

6) Narrow set of solutions once s_i has been found. The new set M_i is calculated as:-

$$M_i \leftarrow \bigcup_{(a,b,r)} \left\{ \left[\max\left(a, \left\lceil \frac{2B + rn}{s_i} \right\rceil\right), \min\left(b, \left\lfloor \frac{3B - 1 + rn}{s_i} \right\rfloor\right) \right] \right\}$$

$$\text{for all } [a, b] \in M_{i-1} \text{ and } \frac{as_i - 3B + 1}{n} \leq r \leq \frac{bs_i - 2B}{n}.$$

7) Finally, we compute solution if M_i has only one interval of length 1 i.e., $M_{i-1} = [a, a]$. Then we let $m = a.(s_0)^{-1} modn$, and return m as solution. If $M_{i-1} = [a, b]$, then we set *i* = *i* + *1* and go to Step 5.

As described, the attack requires only one ciphertext and oracle for querying PKCS compliance of ciphertexts. For a real-world scenario, if the error messages from any software implementing PKCS returns verbose errors, it can be exploited. For this purpose, its best if error messages aren't returned. Another option is using OAEP instead of PKCS. OAEP doesn't have any known attacks and it is considered IND-CCA2 [8] secure.

IV. Optimal Asymmetric Encryption Padding (OAEP)

Optimal Asymmetric Encryption padding (OAEP) is one of the Padding Schemes which is employed with RSA. It was officially published in RFC2437 [9] in October 1998. OAEP is the preferred padding scheme since it is considered to be IND-CCA2 secure. However, there have been studies which contradict this claim stating OAEP is only IND-CCA1 secure.

In comparison to PKCS#1 v1.5, OAEP also incorporates the use of a Hash function hash() and Mask Generation Function (MGF) mask(). Output length of the hash function is represented as hlen bytes.

OAEP also uses an optional label for each message. For this project, we consider the label to always be an empty string. As with PKCS#1 v1.5, the input message is encoded using OAEP before encryption on the sender's end. For a k- byte modulus RSA, the encoding process is as follows (Fig. 4):

1) Length checking message(m): If *len(m)* > *k* - 2 - 2 * *hlen*, report error and stop.
2) Create data block (*DB*):
   a) Append *Hash(L)* as start of string.
   b) Append (*k* - *len(m)* - 2 * *hlen* - 2) *0x00* bytes.
   c) Append *0x01* as separation byte to mark end of *0x00* bytes. Append message to form *DB* of length (*k* - *hlen* - 1) bytes.
3) Generate random seed (*s*) where *len(s)* = *hlen* bytes.
4) Generate *DBmask* = *mask(s, k - hlen - 1)*
5) Generate *maskedDB* = *DBmask* $\oplus$ *DB*
6) Generate *seedMask* = *mask(maskedDB, hlen)*
7) Generate *maskedSeed* = *seedMask* $\oplus$ *seed*
8) Generate Encoded message (*EM*) as:-

$$EM = 0x00||maskedSeed||maskedDB$$

The MGF uses hash() along with a counter to produce an output string os of given length n for a given input string is. This can be represented as (Fig. **??**):

$$mask(is, n) = op, len(op) = n$$

The encoded message is then ecnrypted with the public-key pair using RSA. The generated ciphertext is transmitted to the reciever who then decrypts this using the private-key pair for

RSA. Since there are no known attacks against OAEP, the message structure isnt validated. Instead, before decryption, a basic length check for the ciphertext (*c*) is performed:- if *len(c) != k* bytes OR *len(c)* $<$ *2hLen + 2*, the ciphertext is discarded. Else, it is decrypted to acquire the encoded message (EM). The decoding process for retrieving the plaintext from EM is as follows (Fig. 4):-

1) If *len(EM) != k* bytes OR *len(EM)* $<$ *2hLen + 2*, report error and stop.
2) Split *EM* as *0x00* || *maskedSeed* || *maskedDB*.
3) Retrieve Data Block (*DB*):-
   a) Let *seedMask = mask(maskedDB, hlen)*
   b) Let *seed = seedMask* $\oplus$ *maskedSeed*
   c) Let *DBmask = mask(s, k - hLen - 1)*
   d) Let *DB = DBmask* $\oplus$ *maskedDB*
4) Split *DB* to retrieve original plaintext:-
   a) Discard *Hash(L)* from the data block.
   b) Discard *0x00* bytes until separation byte *0x01* is encountered. If *0x01* is not encountered, report error and stop.
   c) Discard *0x01* and recover plaintext.

V. SHA-3 and SHAKE128/256 with OAEP

The FIPS-186 5 Draft [6] recommends usage of SHA-3/Keccak [10] as valid hash functions for OAEP. This does not affect the computational performance for OAEP drastically, however it does introduce greater security. The main advantage of SHA-3 over its predecessors is that it is not vulnerable to a length extension attack.

The Keccak family of hash functions also introduce SHAKE128/256 [10] which produce variable length hashed output. Since, SHAKE128/256 produces variable length output

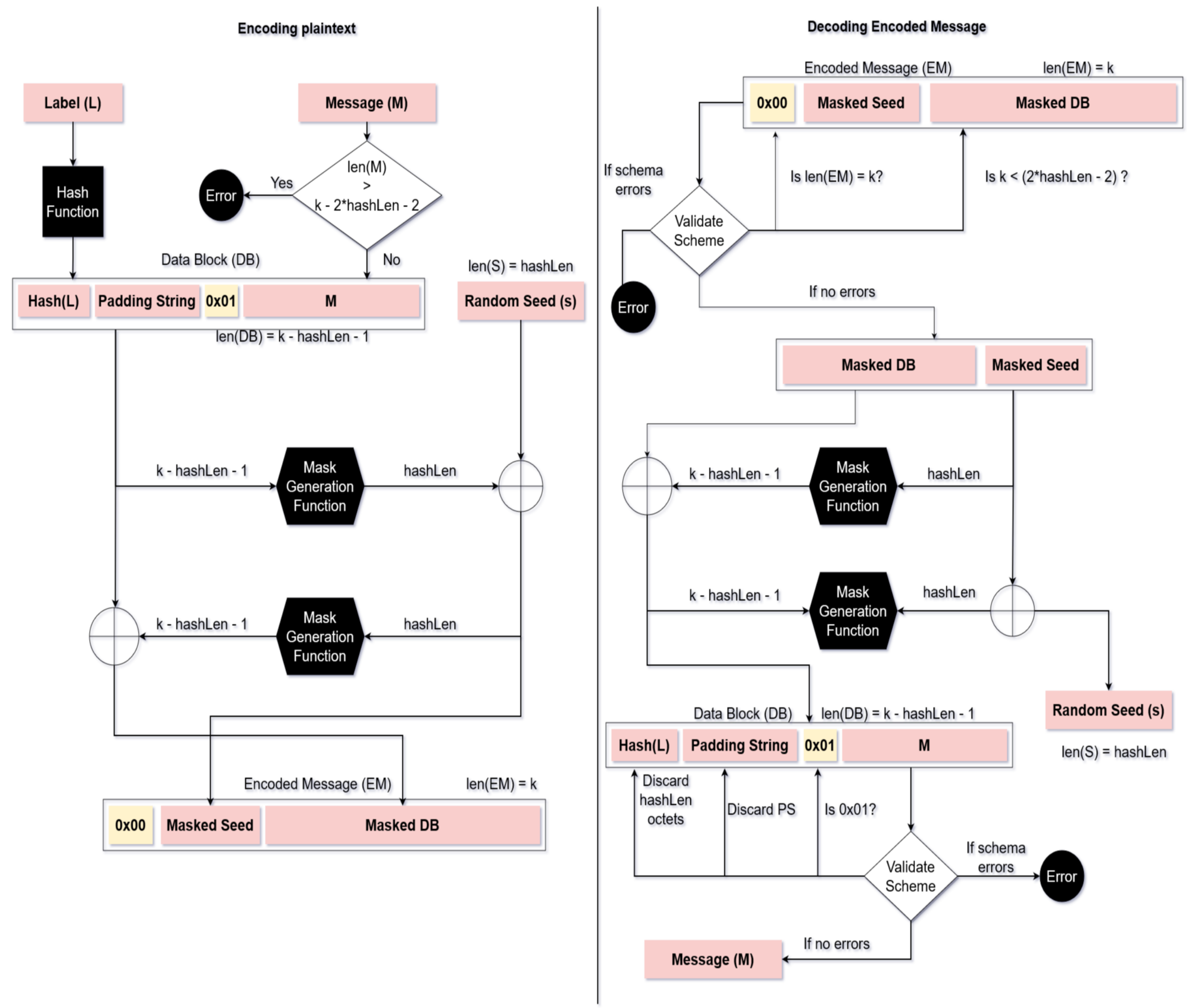

Fig. 4. Encoding and Decoding with OAEP

it can be used as a Mask Generation Function (MGF) [6]. Using SHAKE128/256 as a mask generation function is as [6]: instead of *DBmask = mask(s, k - hlen - 1)*, then either DBmask = *SHAKE128(s, 8*(k - hlen - 1))* or *SHAKE256(s, 8*(k - hlen - 1))*.

Using SHAKE128/256 as a hash function, the output lengths are then fixed at 256/512 bits respectively. However, this paper proposes modifications to OAEP when using SHAKE128/256. The encryption part of RSA-OAEP is modified as follows (Fig. 5):-

1) Randomly generate hash length (hlen) in bytes as follows:-
   a) For SHAKE128: 8 $\leq$ hlen $\leq$ 32
   b) For SHAKE256: 32 $\leq$ hlen $\leq$ 64
2) Length checking message(m): If *len(m)* $>$ *k* - 3 - 2 * *hlen*, report error and stop.
3) Create data block (*DB*):
   a) Append *Hash(L)* as start of string.
   b) Append (*k* - *len(m)* - 2 * *hlen* - 3) *0x00* bytes.
   c) Append *0x01* as separation byte to mark end of *0x00* bytes. Append message to form *DB* of length (*k* - *hlen* - 2) bytes.
4) Generate random seed (*s*) where *len(s)* = *hlen* bytes.
5) Generate *DBmask* = *mask(s, k - hlen - 2)*
6) Generate *maskedDB* = *DBmask* $\oplus$ *DB*
7) Generate *seedMask* = *mask(maskedDB, hlen)*
8) Generate *maskedSeed* = *seedMask* $\oplus$ *seed*
9) Generate Encoded message (*EM*) as:-

$$EM = 0x00||hex(hlen)||maskedSeed||maskedDB$$

The extra byte which contains the hex representation of the byte length is used to split the encoded message while

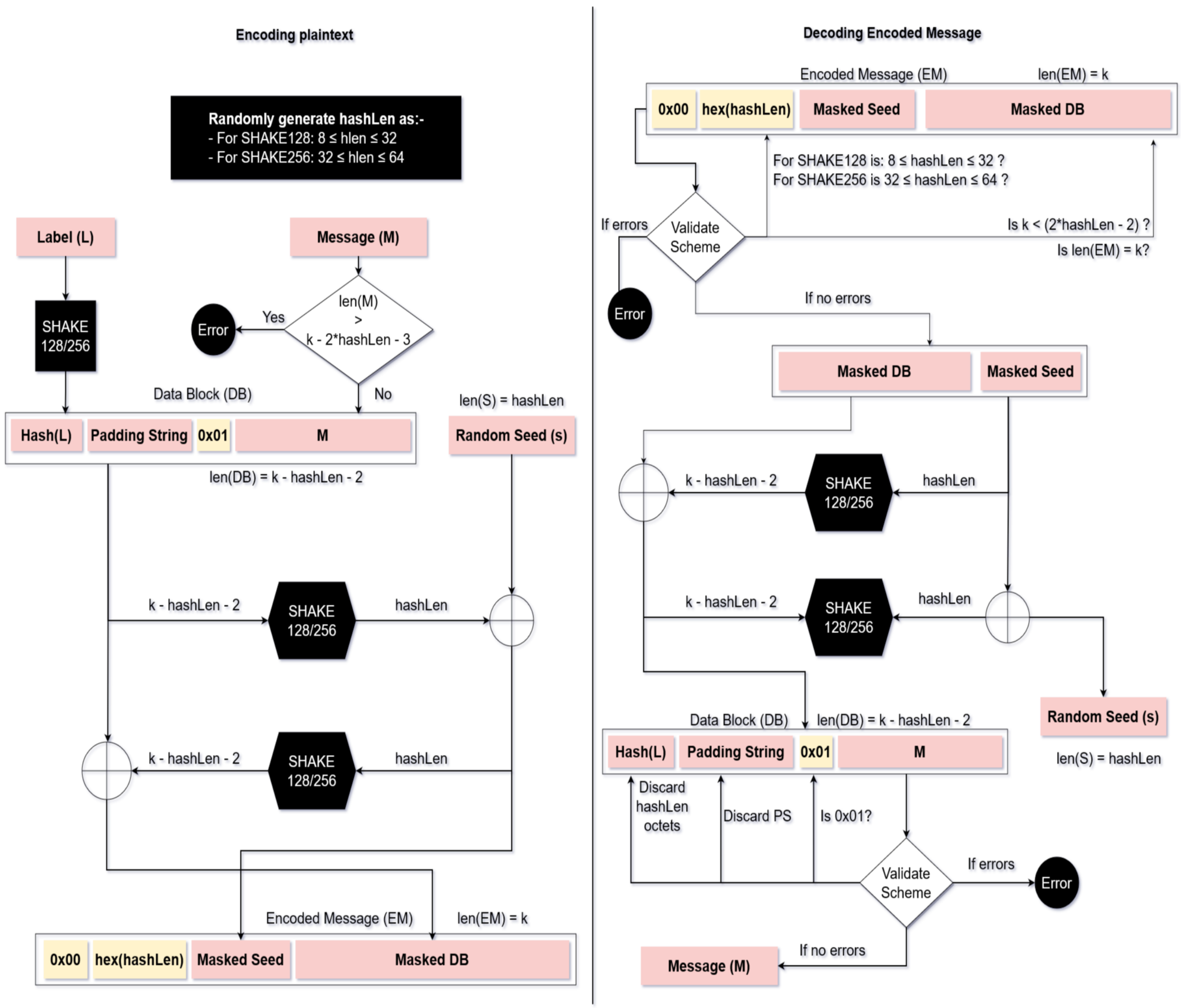

Fig. 5. Encoding and Decoding with modified OAEP scheme

decoding. The decryption for the updated scheme is as follows (Fig. 5):-

1) Check 2^{nd} byte for hlen and validate as follows:-
   a) For SHAKE128 if 8 $<$ hlen or hlen $>$ 32, report error and stop.
   b) For SHAKE256 if 32 $<$ hlen or hlen $>$ 64, report error and stop.
2) If *len(EM) != k* bytes OR *len(EM)* $<$ *2hLen + 3*, report error and stop.
3) Split *EM* as *0x00* || *hex(hlen)* || *maskedSeed* || *maskedDB*.
4) Retrieve Data Block (*DB*):-
   a) Let *seedMask = mask(maskedDB, hlen)*
   b) Let *seed = seedMask* $\oplus$ *maskedSeed*
   c) Let *DBmask = mask(s, k - hLen - 2)*
   d) Let *DB = DBmask* $\oplus$ *maskedDB*
5) Split *DB* to retrieve original plaintext:-
   a) Discard *hlen* bytes from the data block.
   b) Discard *0x00* bytes until separation byte *0x01* is encountered. If *0x01* is not encountered, report error and stop.
   c) Discard *0x01* and recover plaintext.

VI. Padding Schemes for Data Transmission

Since RSA is computationally expensive, it is used for Key encapsulation mechanism (KEM) and Digital Signatures. In the real world RSA is implemented with 2048/4096-bit numbers. In theory this means RSA can be used for transmission of upto 2kb/4kb data in case of 2048-bit/4096-bit RSA respectively. However, RSA is always employed with padding schemes and this reduces the effective message length.

TABLE I
Maximum message length for PKCS#1 v1.5

RSA Modulus (bits)	RSA Modulus (bytes)	Max. message length (bytes)
1024	128	117
1294	161	150
1536	192	181
1626	203	192
2048	256	245
4096	512	501

TABLE II
Maximum message length for OAEP with SHA-224

RSA Modulus (bits)	RSA Modulus (bytes)	Max. message length (bytes)
1024	128	70
1294	161	103
1536	192	134
1626	203	145
2048	256	198
4096	512	454

As part of KEM, RSA is used for secure transmission of keys for Symmetric key algorithm. One of the widely used Symmetric key algorithms is Advanced Encryption Standard (AES) [11]. AES keys can be of variable lengths which are 128-bit (16 bytes), 192-bit (24 bytes) and 256-bit (32 bytes). The tables show maximum message lengths when using various bit length RSA with various padding schemes. As seen, the key lengths for AES are well within the maximum lengths for most padding schemes.

TABLE III
Maximum message length for OAEP with SHA-256

RSA Modulus (bits)	RSA Modulus (bytes)	Max. message length (bytes)
1024	128	62
1294	161	95
1536	192	126
1626	203	137
2048	256	190
4096	512	446

TABLE IV
Maximum message length for OAEP with SHA-384

RSA Modulus (bits)	RSA Modulus (bytes)	Max. message length (bytes)
1024	128	30
1294	161	63
1536	192	94
1626	203	105
2048	256	158
4096	512	414

TABLE V
Maximum message length for OAEP with SHA-512

RSA Modulus (bits)	RSA Modulus (bytes)	Max. message length (bytes)
1024	128	NA
1294	161	31
1536	192	62
1626	203	73
2048	256	126
4096	512	382

When using RSA for Digital Signatures, input message is hashed and encrypted for authentication of the sender. Commonly used Hash functions are SHA-256 (output = 256 bits/32 bytes), SHA-512 (output = 512 bits/64 bytes). Except tables IV, V SHA-256 as an input to RSA in most cases. Also, SHA-512 can be used in most cases when using RSA

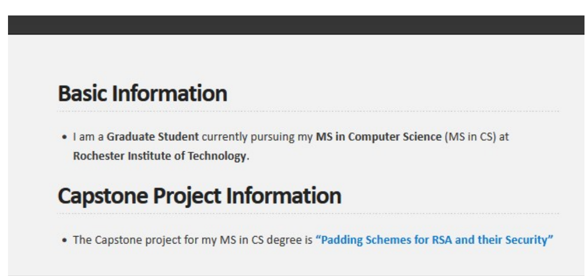

Fig. 6. Basic web page, size = 4.71 kb

modulus is larger. The message length is more restricted when using OAEP as compared to PKCS. This is because OAEP incorporates the hash for message label. The tables compare messages lengths with OAEP when used with SHA3 counterparts.

The maximum message lengths for padding barely reach 0.5 KB when using OAEP (table II). In case of PKCS, the maximum message length turns out to be 0.5 KB. When sending data over the internet, Maximum Transmission Unit value is usually 1.5 KB. If we were to use RSA-4096 with PKCS as a substitute for AES with HTTPS/2 [12] the maximum data we can transmit would be 0.5 KB. Web pages encrypted and sent over the internet via HTTPS/2 [12] are usually at-least 2 KB. As seen in Fig. 6, a simple web-page with just 2 headings and minimal styling is 4.71 KB.

VII. Results for Prime Number Generation

The initial prime number generation algorithm was implemented as follows:-

1) Generate seed value for random bit generation
2) Generate random bits for given bit length
3) Set least and most significant bit to 1, thus generating large odd number
4) Check if number is prime using following primality tests:
    a) Miller-Rabin [13] primality test.
    b) Lucas-Lehmer [14] primality test.
5) If number is prime record it, else check for next number.
6) If safe prime check is enabled, check if recorded prime number is safe prime and save.
7) If safe prime check is not enabled, save recorded prime.

The above algorithm was fast in case of basic primes but very poor when it came to safe primes. Directly generating safe primes turned out to be computationally expensive. This is because randomly generating a prime and checking for it being safe in the same loop.

Thus, the decision to split generation of safe primes as a filtering process was chosen. In this case, a bunch of basic prime numbers are generated and then safe primes are filtered out of them. This created a significant performance increase. Since higher bit numbers have lower rounds in primality tests, a filtering process was chosen. The filtering process checks the odd number for factors within $3..2^{20}-1$. If any factors are present, the odd number is discarded from primality tests. This affected the performance of the new algorithm but not by much. Due to the filtering process, more odd numbers are filtered but better quality of odd numbers are chosen for primality tests. The new prime number selection algorithm is as follows:

1) Generate seed value for random bit generation
2) Generate random bits for given bit length
3) Set least and most significant bit to 1, thus generating large odd number
4) Check if number has factors within 1st million numbers, if yes discard and check for next number.
5) Check if number is prime using following primality tests:
    a) Miller-Rabin [13] primality test.
    b) Lucas-Lehmer [14] primality test.
6) If number is prime save it, else check for next number.
7) Once all primes are generated, segregate primes as safe and basic primes.

The execution statistics for both algorithms are shown in Table VI. The times are when prime numbers for variable lengths are generated together. The generation times for safe primes of higher bit length values were poorer than below execution times.

TABLE VI
Execution statistics for algorithms

Prime	Algorithm	No. of primes	Time taken
Basic	Old	10	3 secs.
Safe	Old	3	6 mins.
Basic	Current	5	8 mins.

VIII. Conclusion

Apart from being deterministic, RSA is also susceptible to hardware attacks [15]. Apart from timing attacks, [15] also discusses broadcast attacks, common modulus attacks which are not discussed by the paper. Some security aspects discussed in [16] [17] [18] were considered while implementing RSA for experiments. Since the RSA tests were performed on a personal laptop guidelines listed in [2] and [6] were taken into consideration. Thus, the RSA implementations are considered secure.

This paper explores the security of padding schemes employed with RSA. PKCS #1 v1.5 is susceptible to Bliechenbacher's attack. The attack is easy to implement and execute given an oracle which validates ciphertext. The error messages from APIs implementing PKCS can be treated as random oracles for checking ciphertexts. Not returning verbose error messages or having security checks for number of failed requests are ways of mitigating these attacks. The best solution is to use OAEP instead of PKCS.

OAEP is the best choice for using RSA in real-world applications. There are no mathematical attacks known against OAEP. However, there have been discussions regarding if OAEP is IND-CCA2 security [19] [20]. The paper does not explore this aspect of OAEP. The paper concludes that OAEP is the best choice not taking into consideration these discussions.

Using RSA with padding schemes for data transmission doesn't work out using current real-world implementations. Using RSA for data transmission, would require higher bit-lengths and this would further increase computational costs. Symmetric key algorithms such as AES [11] with Cipher Block Chaining (CBC) is a better suited option.

To conclude, RSA is currently best used for Key Encryption Mechanism (KEM) since these are one time operations. Elliptic Curve Digital Signature Algorithm (ECDSA) [21] is gaining popularity over RSA since it offers strong security for lower bit-length keys. However, RSA is still a viable option for digital signature algorithms.

References

[1] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," in *Secure communications and asymmetric cryptosystems*. Routledge, 2019, pp. 217–239.

[2] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "PKCS# 1: RSA Cryptography Specifications Version 2.2," *Internet Engineering Task Force, Request for Comments*, vol. 8017, p. 72, 2016.

[3] C. F. Kerry and C. R. Director, "FIPS PUB 186-4 Federal Information Processing Standards publication Digital Signature Standard (DSS)," 2013.

[4] L. M. Adleman and K. S. McCurley, "Open problems in Number Theoretic Complexity, II," in *International Algorithmic Number Theory Symposium*. Springer, 1994, pp. 291–322.

[5] D. Bleichenbacher, "Chosen Ciphertext Attacks against protocols based on the RSA encryption standard PKCS# 1," in *Annual International Cryptology Conference*. Springer, 1998, pp. 1–12.

[6] N. I. of Standards and Technology, "FIPS PUB 186-5 (draft) Federal Information Processing Standards publication Digital Signature Standard (DSS)," 2013.

[7] B. Kaliski, "PKCS# 1: RSA Encryption Version 1.5," RFC 2313, March, Tech. Rep., 1998.

[8] D. Dolev, C. Dwork, and M. Naor, "Nonmalleable cryptography," *SIAM review*, vol. 45, no. 4, pp. 727–784, 2003.

[9] B. Kaliski and J. Staddon, "PKCS# 1: RSA Cryptography Specifications Version 2.0," 1998.

[10] M. J. Dworkin *et al.*, "SHA-3 standard: Permutation-based hash and extendable-output functions," 2015.

[11] J. Daemen and V. Rijmen, "Reijndael: The Advanced Encryption Standard." *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, vol. 26, no. 3, pp. 137–139, 2001.

[12] M. Belshe, R. Peon, and M. Thomson, "Rfc 7540: Hypertext Transfer Protocol Version 2 (http/2)," *Internet Engineering Task Force (IETF)*, 2015.

[13] R. Pavuluru, "Miller-Rabin," 2015.

[14] D. Cortild and A. V. Sanabria, "Lucas-Lehmer Primality Test."

[15] D. Boneh *et al.*, "Twenty years of attacks on the RSA cryptosystem," *Notices of the AMS*, vol. 46, no. 2, pp. 203–213, 1999.

[16] M. J. Wiener, "Cryptanalysis of short RSA secret exponents," *IEEE Transactions on Information theory*, vol. 36, no. 3, pp. 553–558, 1990.

[17] M. Calderbank, "The RSA cryptosystem: history, algorithm, primes," *Chicago: math. uchicago. edu*, 2007.

[18] F. J. Aufa, A. Affandi *et al.*, "Security system analysis in combination method: RSA encryption and digital signature algorithm," in *2018 4th International Conference on Science and Technology (ICST)*. IEEE, 2018, pp. 1–5.

[19] V. Shoup, "OAEP reconsidered," in *Annual International Cryptology Conference*. Springer, 2001, pp. 239–259.

[20] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern, "RSA-OAEP is secure under the rsa assumption," in *Annual International Cryptology Conference*. Springer, 2001, pp. 260–274.

[21] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.