# 프로젝트 #5

소프트웨어학부 암호학

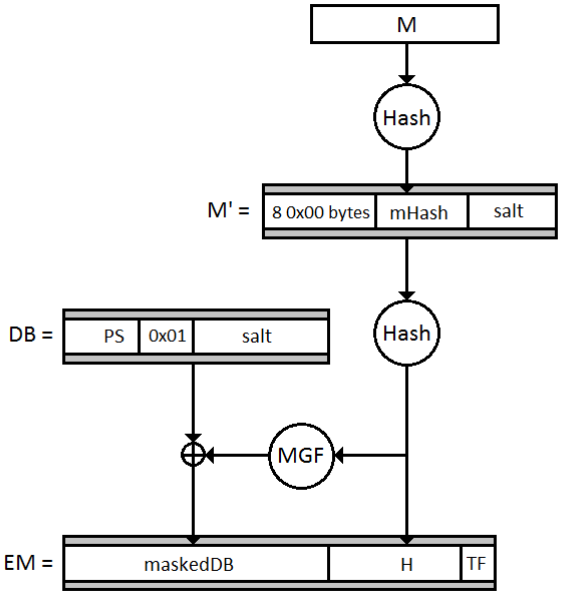
2021년 11월 18일

## 목표

키의 길이가 2048 비트인 RSA-PSS 확률적 전자서명 기법을 구현한다. RSA-PSS는 RSA 공개키 알고리즘을 기반으로 표준화된 서명기법으로 RSASSA-PSS (RSA Probabilistic Signature Scheme with Appendix)라는 긴 이름으로 표기하기도 한다.

## RSA-PSS

RSA-PSS 기법은 서명할 메시지 M을 아래 그림과 같은 과정을 거쳐 EM으로 변환한 후, 개인키 $(d, n)$을 사용하여 $\text{EM}^d \bmod n$을 계산한다.




- 해시함수는 길이가 최소 224 비트인 SHA-2 계열의 함수를 사용한다.
- 난수 salt의 길이는 해시함수의 길이와 같이 한다.
- M'의 처음 8 바이트는 0x00으로 채운다.
- PS는 길이에 맞춰 0x00으로 채운다.
- TF는 1 바이트이며 0xBC로 채운다.
- EM의 길이는 RSA 키의 길이인 `RSAKEYSIZE` (2048 비트)와 일치해야 한다.
- EM의 가장 왼쪽 비트 (MSB)가 1이면 강제로 0으로 바꾼다.

## 전역 함수

외부에서 보이는 전역 함수를 아래 열거한 프로토타입을 사용하여 구현한다. 이번 프로젝트에서는 아래 열거된 함수 중에서 `rsassa_pss_sign()`과 `rsassa_pss_verify()`만 작성하면 된다. 각 함수에 대한 요구사항은 다음과 같다.

- `void rsa_generate_key(void *e, void *d, void *n, int mode);`
  – 길이가 `RSAKEYSIZE`인 `e`, `d`, `n`을 생성한다. `mode`가 0이면 표준 모드로 `e` = 65537을 선택하고, 0이 아니면 무작위로 선택한다. 이 함수는 기본으로 제공한다.

- `void sha224(const unsigned char *m, unsigned int len, unsigned char *digest);`
  `void sha256(const unsigned char *m, unsigned int len, unsigned char *digest);`
  `void sha384(const unsigned char *m, unsigned int len, unsigned char *digest);`
  `void sha512(const unsigned char *m, unsigned int len, unsigned char *digest);`
  – 길이가 `len` 바이트인 메시지 `m`의 SHA-2 해시 값을 `digest`에 저장한다. 이 함수군은 오픈 소스로 기본으로 제공한다.

- `int rsassa_pss_sign(const void *m, size_t mLen, const void *d, const void *n, void *s);`
  – 길이가 `mLen` 바이트인 메시지 `m`을 개인키 `(d,n)`으로 서명한 결과를 `s`에 저장한다. 성공하면 0, 그렇지 않으면 오류 코드를 넘겨준다. `s`의 크기는 `RSAKEYSIZE`와 같아야 한다.

- `int rsassa_pss_verify(const void *m, size_t mLen, const void *e, const void *n, const void *s);`
  – 길이가 `mLen` 바이트인 메시지 `m`에 대한 서명이 `s`가 맞는지 공개키 `(e,n)`으로 검증한다. 성공하면 0, 그렇지 않으면 오류 코드를 넘겨준다.

## 지역 함수

내부에서만 사용하는 지역 함수로 별도의 작성이 필요 없으며 기본으로 제공한다.

- `static int rsa_cipher(void *m, const void *k, const void *n);`
  – $m \leftarrow m^k \bmod n$을 계산한다. 성공하면 0, 그렇지 않으면 오류 코드를 넘겨준다.

- `static unsigned char *mgf(const unsigned char *mgfSeed, size_t seedLen, unsigned char *mask, size_t maskLen);`
  – 길이가 `seedLen` 바이트인 `mgfSeed`를 입력값으로 사용하여 길이가 `maskLen` 바이트인 랜덤 마스크 값을 생성하여 `mask`에 저장한다. 성공하면 마스크의 주소를, 그렇지 않으면 NULL을 넘겨준다.

## 오류 코드

서명 생성과 검증 과정에서 발생하는 오류를 아래에 열거한 코드를 사용하여 식별한다.

- EM_MSG_OUT_OF_RANGE – RSA 데이터 값이 모듈러스 $n$ 보다 크거나 같음
- EM_MSG_TOO_LONG – 해시함수의 입력 데이터가 너무 길어 한도를 초과함
- EM_HASH_TOO_LONG – 해시의 길이가 너무 커서 EM에 수용할 수 없음
- EM_INVALID_LAST – EM의 마지막 (LSB) 바이트가 0xBC가 아님
- EM_INVALID_INIT – EM의 처음 (MSB) 비트가 0이 아님
- EM_INVALID_PD2 – DB의 앞 부분이 0x0000..00||0x01과 일치하지 않음
- EM_HASH_MISMATCH – 해시 값이 일치하지 않음

## 테스트 벡터

다음은 RSA 키의 길이가 1024 비트이고 SHA-224를 사용해서 생성한 검증 벡터이다. 아래 벡터를 사용하여 프로그램이 올바르게 돌아가는지 확인하고 난 후, 키의 길이를 2048로 확장하여 구현한다.

```
RSA key pairs with size = 1024

e = 0000000000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000010001

d = 06c3eeac459356f743956e0810a0d3f9142943bcce19d111de0e5297aed166b4
    369b3b3e15c9479bf48392b34f71923b3569777e4f05295137518f13e6f48fde
    bd6ebf84a5d8ced584848b8a764ae480506f7d4a5ef7fb7cd5fdf8bf89e5aa45
    e39b99af2beda7fdf43261634ddae38f5beffd0bebf9bd51bfe401d5f6d68671

n = c7b8ab9cdb83fbb008d80e78b2265aa088cdb5f9c11c0a92948c4c56e138730a
    4c815dc9b096fe4c1f4fb5259c0209c6c330ff8349bd9e0687ee49824f63f551
    414795733bdee587b4d3efecda10b2baaf0666458b5d21fd2b975a1babe9305d
    3ac28bed9037a4dab14ce9c414a96ebb412e8d26d6e69610191b3bed82e42dc1

With SHA-224, message = "sample"

mHash = 9003e374bc726550c2c289447fd0533160f875709386dfa377bfd41c
salt  = 6e41978602acca182e8bf511b9acdb04ab324572358c153de6cd3e0a

M'  = 0000000000000000009003e374bc726550c2c289447fd0533160f875709386dfa3
      77bfd41c6e41978602acca182e8bf511b9acdb04ab324572358c153de6cd3e0a

H   = de074849895435205639e196634c62305d6433373f688dd840b07f6b

DB  = 0000000000000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000000000000
      000000000000016e41978602acca182e8bf511b9acdb04ab324572358c153de6
      cd3e0a

MGF = c93c72a5e3780e32843ab05bfeef5bc18a78688c859081c546bd86ed2895ce14
      028c98ae4df01c28a6d00b29d7bd9d919b2e28fe95ab09bb9db7c688813627ad
      26d0ee008824235a1ab836231651d93b4e3261c925b5a61e2957c54ae44d00d8
      b84f1f

EM  = 493c72a5e3780e32843ab05bfeef5bc18a78688c859081c546bd86ed2895ce14
      028c98ae4df01c28a6d00b29d7bd9d919b2e28fe95ab09bb9db7c688813627ad
      26d0ee00882422345b2fb021ba9bc115c5c77070896ea2b51b12b77f68583d3e
      757115de074849895435205639e196634c62305d6433373f688dd840b07f6bbc

sig = 80dbbc4987617db7bf5f07f85078bdea501588eca3525dd69023926340e57125
      ec4443d54632acd4c97895394bcf7242fc38a57ce40243783e9a97b3d1eac45f
      8256aa520e44d77120a0585db14c3f4d5195058014e6d924092e57de6237c405
      88b08a6cdc1b73b5fbed022ea12470993dfd3c01480f841863eb23d0383f1b8f
```

## GNU GMP 라이브러리 설치

GNU GMP 라이브러리는 정수의 크기가 $2^{64}$보다 큰 수를 계산하기 위해 개발된 패키지이다. 프로젝트에서 기본적으로 제공하는 `rsa_generate_key()`와 `rsa_cipher()` 함수는 GMP 라이브러리를 사용하고 있기 때문에 이 함수를 활용하려면 각자 환경에 맞는 GMP 라이브러리를 설치해야 한다. GMP는 Linux, MacOS, Windows 등 대부분의 환경을 지원한다. 인터넷에서 방법을 찾아서 먼저 설치하고 프로젝트를 진행한다.

## 골격 파일

구현에 필요한 골격파일 `rsa_pss.c`와 함께 헤더파일 `rsa_pss.h`, 프로그램을 검증할 수 있는 `test.c`, SHA-2 오픈소스 `sha2.c`, `sha2.h` 그리고 `Makefile`을 제공한다. 이 가운데 `test.c`, `sha2.c`, `sha2.h`를 제외한 나머지 파일은 용도에 맞게 자유롭게 수정할 수 있다.

## 제출물

과제에서 요구하는 함수가 잘 설계되고 구현되었다는 것을 보여주는 자료를 보고서 형식으로 작성한 후 PDF로 변환하여 `이름_학번_PROJ5.pdf`로 제출한다. 여기에는 다음과 같은 것이 반드시 포함되어야 한다.

- 본인이 작성한 함수에 대한 설명
- 컴파일 과정을 보여주는 화면 캡처
- 실행 결과물의 주요 장면과 그에 대한 설명, 소감, 문제점 등
- 프로그램 소스파일 (`rsa_pss.c`, `rsa_pss.h`) 별도 제출
- 프로그램 실행 결과 (`rsa_pss.txt`) 별도 제출

## 평가

- Correctness 50%: 프로그램이 올바르게 동작하는 지를 보는 것입니다. 여기에는 컴파일 과정은 물론, 과제가 요구하는 기능이 문제없이 잘 작동한다는 것을 보여주어야 합니다.
- Presentation 50%: 자신의 생각과 작성한 프로그램을 다른 사람이 쉽게 이해할 수 있도록 프로그램 내에 적절한 주석을 다는 행위와 같이 자신의 결과를 잘 표현하는 것입니다. 뿐만 아니라, 프로그램의 가독성, 효율성, 확장성, 일관성, 모듈화 등도 여기에 해당합니다. 이 부분은 상당히 주관적이지만 그러면서도 중요한 부분입니다. 컴퓨터과학에서 중요하게 생각하는 best coding practices를 참조하기 바랍니다.

*HK*