

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ  
СІКОРСЬКОГО»  
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО ТЕХНІЧНИЙ ІНСТИТУТ  
КАФЕДРА МАТЕМАТИЧНОГО МОДЕЛЮВАННЯ ТА АНАЛІЗУ  
ДАНИХ**

Лабораторна робота №2  
З дисципліни «Методи реалізації криптографічних механізмів»

Студента групи ФІ-21мн  
Прохоренко О.С.

Викладач:  
Селюх П.В.

Київ-2023

# ХІД РОБОТИ

## 1. Завдання

Аналіз стійкості реалізацій ПВЧ та генераторів ключів для обраної бібліотеки.  
Підгрупа 2В. Бібліотека PyCrypto під Linux платформу.

## 2. Результати

**PyCrypto** давно не підтримується та має дірки в безпеці, рекомендують використовувати **cryptography** або **PyCryptodome**, але саме **PyCryptodome** є форком **PyCrypto**.  
PyCryptodome - це розширена і більш безпечна версія PyCrypto, яка включає у себе різноманітні криптографічні інструменти.

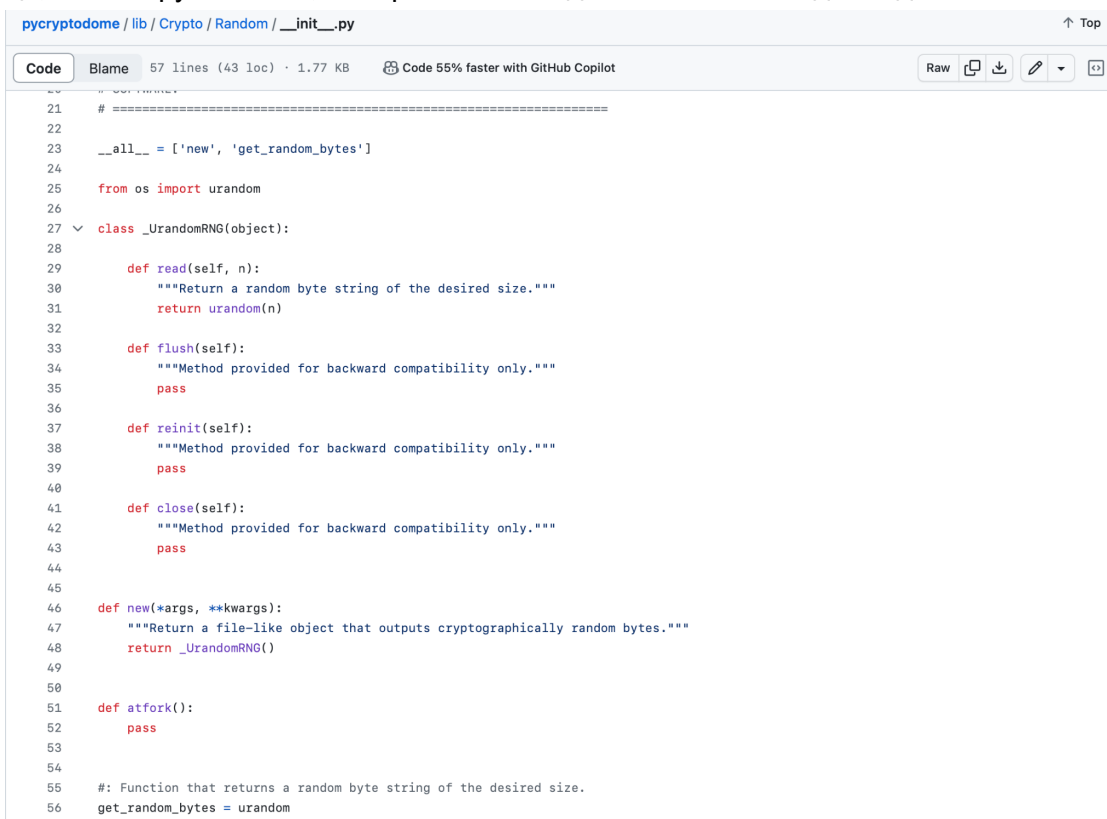
### Генерація ПВП (Псевдовипадкових Послідовностей)

PyCryptodome використовує таке джерело ентропії для генерації ПВП, як системні джерела випадковості, такі як /dev/urandom у Linux або функція CryptGenRandom для Windows.

Вхідних даних як таких немає.

Вихідні дані - це псевдовипадкові байти, які можуть бути використані для різних криптографічних цілей.

Функції, які генерують ПВП, повертають послідовність байтів заданої довжини.



```
pycryptodome / lib / Crypto / Random / __init__.py
Code Blame 57 lines (43 loc) · 1.77 KB Code 55% faster with GitHub Copilot
Raw Copy Download Edit
21 # =====
22
23 __all__ = ['new', 'get_random_bytes']
24
25 from os import urandom
26
27 class _UrandomRNG(object):
28
29     def read(self, n):
30         """Return a random byte string of the desired size."""
31         return urandom(n)
32
33     def flush(self):
34         """Method provided for backward compatibility only."""
35         pass
36
37     def reinit(self):
38         """Method provided for backward compatibility only."""
39         pass
40
41     def close(self):
42         """Method provided for backward compatibility only."""
43         pass
44
45
46     def new(*args, **kwargs):
47         """Return a file-like object that outputs cryptographically random bytes."""
48         return _UrandomRNG()
49
50
51     def atfork():
52         pass
53
54
55 #: Function that returns a random byte string of the desired size.
56 get_random_bytes = urandom
```

```

import chardet
from Crypto.Random import get_random_bytes

# Generation of 16 random bytes
random_bytes = get_random_bytes(16)
result = chardet.detect(random_bytes)
charenc = result['encoding']
print(f"Random bytes: {random_bytes}")
if charenc:
    print(f"Converted bytes to string: {random_bytes.decode(charenc)}")

```

```

Random bytes: b'1\xaeA\xc0\xc7\xdd\xd32\x02\xc6\xfa\x97\xcb\x9f\xa6q'
Converted bytes to string: 1@AÀÇÝÓ2Æú-Ëÿ;q

```

## Генерація Ключів

Генерація ключів у PyCryptodome для асиметричних шифрів, як RSA, включає в себе вибір великих простих чисел з генерацією ПБП на основі функції urandom з os.

Вхідні дані можуть включати довжину ключа.

Вихідні дані - це згенерований криптографічний ключ.

Функції повертають об'єкт ключа.

У разі помилок (наприклад, неправильно задана довжина ключа) повертаються помилки.

[pycryptodome](#) / [lib](#) / [Crypto](#) / [PublicKey](#) / [RSA.py](#)

Code	Blame	833 lines (663 loc) · 29.3 KB	Code 55% faster with GitHub Copilot
427			
428	✓	<code>def generate(bits, randfunc=None, e=65537):</code>	
429		<code>    """Create a new RSA key pair.</code>	
430			
431		<code>    The algorithm closely follows NIST `FIPS 186-4` in its</code>	
432		<code>    sections B.3.1 and B.3.3. The modulus is the product of</code>	
433		<code>    two non-strong probable primes.</code>	
434		<code>    Each prime passes a suitable number of Miller-Rabin tests</code>	
435		<code>    with random bases and a single Lucas test.</code>	
436			
437		<code>    Args:</code>	
438		<code>        bits (integer):</code>	
439		<code>            Key length, or size (in bits) of the RSA modulus.</code>	
440		<code>            It must be at least 1024, but **2048 is recommended.**</code>	
441		<code>            The FIPS standard only defines 1024, 2048 and 3072.</code>	
442		<code>        randfunc (callable):</code>	
443		<code>            Function that returns random bytes.</code>	
444		<code>            The default is :func:`Crypto.Random.get_random_bytes`.</code>	
445		<code>        e (integer):</code>	
446		<code>            Public RSA exponent. It must be an odd positive integer.</code>	
447		<code>            It is typically a small number with very few ones in its</code>	
448		<code>            binary representation.</code>	
449		<code>            The FIPS standard requires the public exponent to be</code>	
450		<code>            at least 65537 (the default).</code>	
451		<code>    """</code>	

pycryptodome / lib / Crypto / PublicKey / RSA.py

Code Blame 833 lines (663 loc) · 29.3 KB Code 55% faster with GitHub C Raw

```

428 def generate(bits, randfunc=None, e=65537):
468     while n.size_in_bits() != bits and d < (1 << (bits // 2)):
469         # Generate the prime factors of n: p and q.
470         # By construction, their product is always
471         # 2^{bits-1} < p*q < 2^bits.
472         size_q = bits // 2
473         size_p = bits - size_q
474
475         min_p = min_q = (Integer(1) << (2 * size_q - 1)).sqrt()
476         if size_q != size_p:
477             min_p = (Integer(1) << (2 * size_p - 1)).sqrt()
478
479         def filter_p(candidate):
480             return candidate > min_p and (candidate - 1).gcd(e) == 1
481
482         p = generate_probable_prime(exact_bits=size_p,
483                                     randfunc=randfunc,
484                                     prime_filter=filter_p)
485
486         min_distance = Integer(1) << (bits // 2 - 100)
487
488         def filter_q(candidate):
489             return (candidate > min_q and
490                     (candidate - 1).gcd(e) == 1 and
491                     abs(candidate - p) > min_distance)
492
493         q = generate_probable_prime(exact_bits=size_q,
494                                     randfunc=randfunc,
495                                     prime_filter=filter_q)
496
497         n = p * q
498         lcm = (p - 1).lcm(q - 1)
499         d = e.inverse(lcm)
500
501         if p > q:
502             p, q = q, p
503
504         u = p.inverse(q)

```

Процес генерації ключів RSA, як описано в коді з PyCryptodome, є наступним:

Алгоритм починає з вибору двох великих простих чисел, які називаються  $p$  і  $q$ . Ці числа повинні бути достатньо великими, щоб їх добуток був занадто великим для розкладання на множники за практичний час, що є основою безпеки RSA.

Кожне з цих простих чисел проходить через серію тестів на простоту, таких як тест Міллера-Рабіна та один тест Лукаса. Тест Міллера-Рабіна - це ймовірнісний тест, який може визначити, чи число є складеним, але не може з абсолютною впевненістю сказати, що число є простим. Тест Лукаса додає додаткову перевірку, яка підвищує стійкість до помилкових позитивних результатів.

Після вибору та перевірки простих чисел  $p$  і  $q$ , алгоритм обчислює модуль  $n = p * q$ , який використовується як частина публічного та приватного ключів. Та обраховується  $d$ .

### 3. Висновок

Узагальнюючи, PyCryptodome вважається безпечною бібліотекою для генерації ключів RSA та ПБП, доки вона правильно використовується і оновлюється.