Deterministic RSA key generation DRSA

Rafael Remígio
DETI University of Aveiro
Aveiro, Portugal

João Almeida
DETI University of Aveiro
Aveiro, Portugal

I. Introduction

The generation of RSA key pairs traditionally involves random generation within the module's dimension. However, an alternative method involves deterministic key pair generation, where a fixed set of parameters consistently generates the same key pair. This deterministic approach ensures predictability in RSA key pair generation.

The *rsagen* tool developed in the scope of this project enables deterministic RSA key pair generation. 3 specific parameters are used to set up the generator: a password, a confusion string, and an iteration count. These ensure the setup of the randomness source used for the RSA key pair generation. Notably, it's crucial to ensure that the setup of a randomness source for the RSA key pair generator is time-consuming to enhance security.

The *randgen* tool evaluates the setup time of the pseudo-random generator mentioned above, focusing on different input parameters to determine their impact on the generation process.

II. rsagen

A. Architecture

The tool operates by accepting three parameters: a password string, a confusion string, and an integer representing the number of iterations. The process unfolds as follows:

1) Derivation of a bootstrap seed by combining the password, confusion string, and iteration count using a password based key derivation function
2) Conversion of the confusion string into a sequence of bytes of equivalent length.
3) Initialization of the generator using the bootstrap seed.
4) Generation of a pseudo-random byte stream using the generator, stopping upon encountering the confusion pattern.
5) Production of a new seed via the generator and utilizing it to reinitialize the generator.
6) Iteration of the previous two steps according to the specified number of iterations.
7) Utilization of the reseeded pseudo-random number generator (PRNG) after the final iteration for RSA key generation purposes.

Algorithm 1 Generating RSA Keys

function GENERATERSA($pwd, confusion, iterations$)
  $seed \leftarrow PBKD(pwd + confusion + iterations)$
  $confusion\_pattern \leftarrow encode(confusion, "utf-8")$
  $prng \leftarrow PseudoRandomGenerator(seed)$
  $count \leftarrow 0$
  $current\_sequence \leftarrow zeros(confusion.length)$
  while $count < iterations$ **do**
    $current\_sequence \leftarrow prng.sequence(confusion\_pattern.length)$
    if $current\_sequence == confusion\_pattern$ **then**
      $count \leftarrow count + 1$
      $new\_seed \leftarrow prng.sequence(seed.length)$
      $prng \leftarrow PseudoRandomGenerator(new\_seed)$
    end if
  end while
  $exponent \leftarrow 2^{16} + 1$
  $rsa\_g \leftarrow RSAGenerator(prng, exponent)$
end function=0

B. Password based key generation

We employ Argon2 for password-based key derivation. This cryptographic method was selected as the winner in the Password Hashing Competition of 2013, aimed at finding new recommended standards for password hash functions.[3][1]

Argon2 stands out for its dual constraints: it is both compute-bound and memory-bound. This characteristic ensures that achieving an equivalent unlock time for users results in a significantly decelerated pace for password cracking attempts. Unlike PBKDF2, Argon2 restricts an attacker from running numerous instances concurrently on a GPU due to memory constraints. Consequently, achieving the same unlock duration for users implies that password cracking becomes exponentially slower compared to PBKDF2 [2]. This attribute aligns perfectly with the fundamental objective of this project.

C. Pseudo random number generator

In adherence to NIST standards for Cryptographically Secure Pseudo Random Number Generation (CSPRNG), our implementation aligns with a version of Hash_DRBG, as specified in NIST SP 800-90A. [4]

This generator orchestrates the production of random bits by applying a hash function iteratively to an initial seed. The Hash_DRBG functions across three primary phases:

1) Instantiate: This phase initializes the internal state of the DRBG by processing the initial seed and any optional additional inputs.
2) Generate: Here, pseudo random bits are generated by iteratively applying the hash function to the current internal state. These bits are extracted and utilized as random output.
3) Reseed: To uphold its security integrity, the DRBG requires periodic reseeding with fresh entropy. Reseeding involves introducing a new seed and optional additional inputs to update the DRBG's internal state. In our case, we cannot introduce entropy outside our control since the pseudo random generation needs to be deterministic, so we simply rely on reseeding the PRNG with the initial seed and an iterative counter.

We used NIST's recommendation for the hashing function within Hash_DRBG, SHA256, ensuring a robust cryptographic foundation.

The decision to utilize Hash_DRBG stemmed from its efficiency, reliance on hashing principles, and relative ease of implementation compared to other cryptographically secure pseudo-random number generators. This selection aligns with the objectives of ensuring both security and practicality within our cryptographic framework.

D. RSA Generation

During the RSA Key generation phase, we replace the default pseudo random number generation function within the library with our customized pseudo random number generator, employing our specified seed after the several iterations through the

Subsequently, we facilitate the export of both the Private and Public keys in PEM format, allowing users to designate the desired file location for storage.

E. Usage

```
usage: rsagen [-h] [-p PASSWORD] [-c
    ↪ CONFUSION] [-i ITERATIONS] [-o
    ↪ PRIVATE] [-f PUBLIC]

Generate deterministic RSA Key pair from a
    ↪  password a confusion string and a
    ↪ number of iterations.

Arguments:
  -h, --help show this help message and
      ↪ exit
  -p PASSWORD, --password PASSWORD
                        Password String used to
                            ↪ generate RSA Key
                            ↪ pair
  -c CONFUSION, --confusion CONFUSION
                        Confusion String used
                            ↪ for iterations on
                            ↪ PRNG
  -i ITERATIONS, --iterations ITERATIONS
                        Number of iterations
                            ↪ that the PRNG
                            ↪ needs to generate
                        confusion
  -o PRIVATE, --private PRIVATE
                        Private key output file
                            ↪ - PEM format
  -f PUBLIC, --public PUBLIC
                        Public key output file -
                            ↪  PEM format
```

III. RANDGEN

A. Architecture

To evaluate the setup time of the described pseudo-random generator under various input parameters, two components were developed. A Python script was created to measure the setup time, calculate the mean for each confusion string at each number of iterations, and generate a graph. Additionally, a corresponding Bash script was designed to provide random data to this Python script.

B. Python Script

The Python script takes three essential arguments: a list of confusion strings, values for the number of iterations, and a parameter defining how many times each confusion string length will be utilized. This enables the calculation of mean values for each confusion string length, resulting in more precise outcomes.

The script initializes the `rsagen` for each confusion string and records the setup time for each specified number of iterations. It then calculates the mean setup time for each confusion string length, considering the mean time across the specified number of iterations.

The script visualizes the mean setup times for different confusion string lengths, with the x-axis representing the confusion string length and the y-axis representing the mean setup time. Additionally, the script saves the resulting plot as 'results.png'.

Due to the time-consuming execution, the script prints the values being calculated and the individual time for each calculation, providing insight into the progress during execution.

C. Bash Script

The Bash script adopts a systematic approach to generate and analyze random confusion strings, followed by the execution of the Python script. Initially, the `generate_confusion_string` function employs the `/dev/urandom` device to create random character sequences of specific lengths, filtering only letters and digits. Command-line parameters are then checked, including the desired number of confusion strings, iteration values, the Python script to be

executed, and an additional parameter, `n`. The script generate a list of confusion strings, allowing for repetition of eac size as specified by `n`. The optional display of these string serves for visualization purposes. Finally, the Python scrip is invoked, passing the confusion strings, iteration value and `n` as arguments, enabling subsequent analysis of th mean setup times concerning different confusion string length and iterations in the associated Python script utilized fo calculating averages and creating the graph (`randgen`).

IV. Results

Analyzing the time required to set up the Pseudo Randor Generator used in generating RSA keys, across 3 differer confusion string lengths and number of iterations.

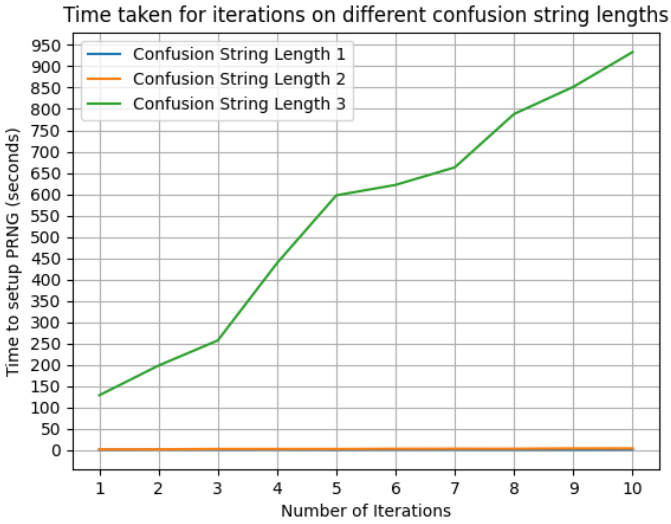

Fig. 1. Time taken in setup on different confusion string lengths and iterations

V. Conclusion

In conclusion, our project has traversed a simple exploration into the intricacies of cryptographic key generation and secure random number generation.

The prolonged setup time associated with initializing the randomness source within our RSA key pair generator, although not pivotal in the key pair creation process, functions as a substantial obstacle for potential attackers. Consequently, our focus centered on employing algorithms imbued with inherent resistance against cracking attempts, such as Argon2 and Hash_DRBG.

References

[1] *Argon2 Repository*. URL: https://github.com/p-h-c/phc-winner-argon2.

[2] George Hatzivasilis. "Password-Hashing Status". In: *Cryptography* 1.2 (2017). ISSN: 2410-387X. DOI: 10.3390/cryptography1020010. URL: https://www.mdpi.com/2410-387X/1/2/10.

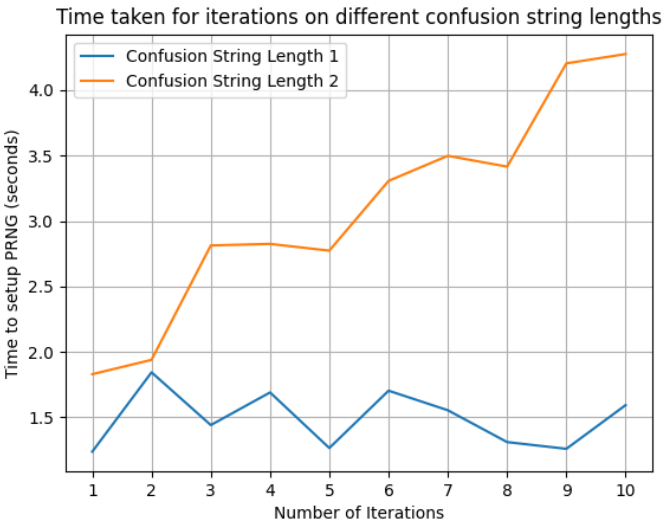

Fig. 2. Time taken in setup on different confusion string lengths and iterations

[3] *Password Hashing Competition*. URL: https://www.password-hashing.net/.

[4] *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. URL: https://doi.org/10.6028/NIST.SP.800-90Ar1.