Analysis of a Cross-Platform Cryptographically Secure Password Generator in C++

Angelos Kamariadis
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
angeloskamariadis@gmail.com

***Abstract*—This paper provides a technical analysis of a C++ application designed to generate secure, random passwords. The software utilizes Operating System (OS) specific cryptographic primitives to ensure high entropy, implements rejection sampling to eliminate modulo bias, and employs the Fisher-Yates shuffle algorithm to ensure uniform distribution of character classes. The code addresses common security pitfalls associated with standard pseudo-random number generators.**

***Index Terms*—cryptography, C++, PRNG, security, modulo bias, rejection sampling**

I. Introduction

Password generation is a critical component of computer security systems. A pervasive vulnerability in many ad-hoc password generators is the reliance on deterministic algorithms, such as the standard C library `rand()`, which can be predicted if the seed state is compromised or guessed.

The code analyzed in this paper implements a **Cryptographically Secure Pseudo-Random Number Generator (CSPRNG)**. It abstracts platform-specific implementations to ensure portability between Windows and POSIX (Linux/Unix) systems, providing a robust solution for secure credential generation.

II. Entropy Collection

The core security feature of the analyzed program is its method of gathering entropy. The function `get_entropy_uint32` serves as a wrapper for OS-level cryptographic functions, ensuring that the random seed is derived from unpredictable hardware and system noise.

A. Cross-Platform Implementation

To maintain portability, the code utilizes preprocessor directives (`#ifdef _WIN32`) to compile specific instructions based on the target operating system:

- **Windows:** It utilizes the Microsoft CryptoAPI (`CryptGenRandom`). This derives randomness from opaque system data such as process IDs, thread IDs, and the high-resolution system clock.
- **Linux/Unix:** It reads from `/dev/urandom`. This is a special character file that provides an interface to the kernel's non-blocking random number generator, gathering environmental noise from device drivers and interrupts.

B. Fallback Mechanism

A fallback function, `fallback_prng`, is implemented using the standard `rand()` seeded with the current time and memory address.

Note: It is crucial to observe that this fallback is **not** cryptographically secure. It serves only as a failsafe to prevent application crashes (Denial of Service) if the OS-level crypto context fails to acquire, though it degrades the security posture significantly.

III. Elimination of Modulo Bias

One of the most sophisticated aspects of this implementation is the `uniform_uint32` function. When mapping a large random number range to a smaller range (e.g., selecting an index for a character array of size 62), a naive approach uses the modulo operator:

$$r \pmod{n} \tag{1}$$

However, if the maximum value of the random number generator ($2^{32}-1$) is not evenly divisible by n, numbers at the lower end of the range will appear slightly more frequently than higher numbers. This statistical anomaly is known as **Modulo Bias**.

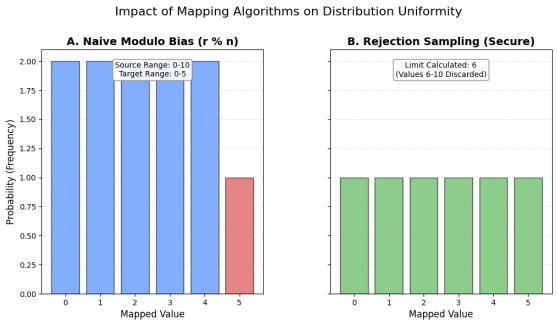

Fig. 1. Visual representation of bias elimination: Naive Modulo vs. Rejection Sampling.

To mitigate this, the code implements **Rejection Sampling**:

1) Calculate a `limit`: the largest multiple of the range that fits within a 32-bit integer.
2) Generate a random number.

3) If the number exceeds this limit, it is "rejected" (discarded), and a new number is generated.

This technique ensures that every possible outcome in the target range has an exactly equal probability of being selected:

$$P(x) = \frac{1}{n}, \quad \forall x \in \{0, \ldots, n-1\} \tag{2}$$

IV. Password Construction Algorithm

The `generate_password` function constructs the password in three distinct phases: Character Filtering, Guaranteed Inclusion, and Backfilling.

A. Pool Filtering

The user's requirements (uppercase, lowercase, digits, symbols) determine the "pools" of allowed characters. The code includes a logic branch for `avoid_ambiguous`. If enabled, characters that look similar (e.g., '1', 'l', 'I', '0', 'O') are mathematically removed from the available pools.

B. Guaranteed Inclusion

To satisfy complexity rules (e.g., "Must contain at least one symbol"), the code iterates through every selected pool and picks one character from each.

$$P_{initial} = \{c_1, c_2, ..., c_k\} \tag{3}$$

Where k is the number of character categories selected. This ensures that the password meets complexity requirements even if the requested length is short.

C. Backfilling and Shuffling

The remaining length of the password is filled by selecting characters uniformly from the combined alphabet of all selected groups.

Because the "Guaranteed Inclusion" step places specific character types at the very beginning of the string (e.g., the first character is always lowercase, the second always uppercase), the resulting string is structurally predictable. To fix this, the code applies a **Fisher-Yates Shuffle**.

This algorithm iterates backward through the string, swapping the current element with a randomly selected element that precedes it. This destroys any structural patterns created during the generation phase.

V. Interface and Input Handling

The `main` function provides a Command Line Interface (CLI). It parses arguments to toggle character sets (e.g., `--no-symbols`) or set length (`-n`).

It is worth noting a minor redundancy in the code: it attempts to read the password size from Standard Input (`cin`) before parsing command-line arguments. This allows the program to be piped into, though the standard CLI convention usually prioritizes arguments over the standard input for configuration.

VI. Conclusion

The analyzed code represents a robust implementation of a password generator. By prioritizing OS-provided entropy over standard library randomization and mathematically correcting for modulo bias, it achieves a high level of cryptographic strength suitable for security-sensitive applications.

References

[1] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Reading, MA: Addison-Wesley, 1997.

[2] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," National Institute of Standards and Technology, Gaithersburg, MD, NIST Special Publication 800-90A Rev. 1, 2015.

[3] R. A. Fisher and F. Yates, *Statistical Tables for Biological, Agricultural and Medical Research*. London: Oliver and Boyd, 1938, pp. 26–27.