A Method of Generating Random Numbers In An Efficient Manner Using Hash Functions

A Hash-Based Pseudo Random Number Generator

Jonathan Alvaro
Teknik Informatika
Institut Teknologi Bandung
Bandung, Indonesia
jonalvarot@gmail.com

Abstract—In computer science, random numbers are used in numerous important applications, such as cryptography and simulations of real-world events. With such a wide array of applications, a method to generate uniformly distributed random numbers in an efficient manner would be very beneficial. This paper explores the possibility of generating uniformly distributed random numbers using hashing algorithms. Initial evaluation shows that this method would be able to generate numbers in a sufficiently uniform manner with low computing power required despite having a limited range of numbers.

Keywords—random; hash; algorithms; uniform.

I. Introduction

Random number generators are being used in various fields of study to perform important tasks, such as cryptography [1] and simulation of complex systems. In cryptography in particular, uniformly distributed random number generators are needed in order to ensure the security of encrypted data. On the other hand, in order for the generator to be applicable, it must be able to generate random numbers without using too much computing power. This paper strives to explore the possibility of using hash algorithms as the method of generating new random numbers. The reason that hash algorithms are chosen is two-fold. First, is the fact that with its widespread use, there are efficient implementations of these algorithms that are available. Second, is due to the way that good hashing algorithms work. Namely, they are designed in such a way that collisions are minimized. Because of this, the author makes an assumption that the hash strings generated by the hashing algorithm could be used as a source of random numbers that are distributed in a sufficiently uniform manner.

II. Background

A. Hashing

Hashing is an operation which maps a value that is given as an input into another completely different value [2]. In general, the output of a hash operation is an array of bits of certain length, with each hashing method having its own length of output. Usually, these output values are displayed as a hex-based string in order to make it more readable by humans.

There are various different methods that could be used to do a hash operation. These methods are usually called hash functions. Each hash functions have their own unique characteristics, such as different output lengths. But, most hash functions that are used in real world settings today usually gives output values that are at least 256 bits long. The reason for this is that hash functions aim to minimize hash collision, which is an event where two different inputs produce the same output values. Hash collisions are undesirable because they cause significant security vulnerabilities [3].

In the real world, there are a lot of important applications for hashing, in particular, in the field of security. For example, a hash string can be used to ensure the integrity of data that is distributed on the web. A user that means to use the data could ensure the data's integrity simply by comparing the hash of the data that he received with the hash that is listed on the data's source. Another example for the use of hash would be to hash account passwords. Since hash algorithms are expected to generate very small amount of collisions, passwords can be passed through a hash function on a client machine, before it is passed on to the server. This way, it ensures that any intercepted packet would not compromise the user account's security.

B. Available Hash Functions

There are a lot of hash functions with various implementations that are readily available for use out there. Amongst them, there is one family of hash functions, the Secure Hash Algorithms (SHA), which is the most widely known. The reason for its popularity is due to the National Institute of Standards and Technology recommending it as the publicly used hash function for security purposes [4]. Among the hash functions that are included within this family, two of them, namely SHA-0 and SHA-1, are considered insecure because of known hash collisions. The other two functions, SHA-2 and SHA-3, are not as compromised as the previous two functions. But, in general, it is recommended that SHA-3 is the used hash function because of SHA-2's known vulnerability against extension attacks.

Another widely known family of hash functions would be the Message Digest (MD) hash functions. There are 4 functions in this family, MD2, MD4, MD5, and MD6. The MD hash functions, more specifically the MD5, used to be a popular hash function due to its speed, until significant vulnerabilities were found. These vulnerabilities presented various attack methods that simply rendered the MD5 unsuitable for security [5]. In response to this, the MD6 algorithm was designed and implemented in order to address the vulnerabilities in MD5. It was in fact, one of the candidates that were considered to be adopted as the SHA-3 algorithm. But, the MD6 was eventually considered not ready for practical applications, mainly due to speed problems.

III. Proposed Algorithm

As the paper title suggests, this paper proposes a method of using various hash functions in order to efficiently generate uniformly distributed random numbers. Due to its simplicity, the proposed algorithm should be compatible with most of the hash functions that are available, thus, making it a very easily implemented algorithm. In general, the algorithm could be divided into two parts, namely:

1. Hash string generation
2. Translation into numbers

The first part is mainly handled by the hash function, which is not the main topic of this paper, and as such, will not be discussed into further details other than what is mentioned in chapter II.

As for the second part, there are numerous methods that are available for this purpose. In general, it would be desirable for the method used in translating the hash string to be able to generate a wide array of numbers in order to increase the applicability of the algorithm in real world applications.

One thing that should be noted here is that a hash string generated by a hash function would be limited in length and, as such, would only be able to generate a limited amount of random numbers. The algorithm solves this problem by using the generated hash string as the input value to generate the next hash string that would be used to generate the numbers. With this method, an initial seed value could be used to generate an unlimited amount of random numbers. Furthermore, considering the fact that good hash functions would rarely encounter a hash collision, it is reasonable to assume that the generated numbers would be distributed in a uniform manner. For a more detailed picture of the algorithm, refer to Fig. 1.

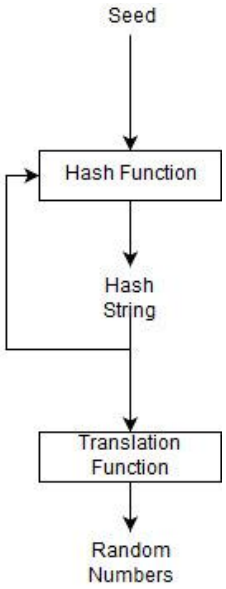

Fig. 1. Flow of Proposed Algorithm

IV. Method

In this paper, the main goal would be to measure the efficiency of the algorithm and the uniformity of the random numbers. In order to measure this, the proposed algorithm would be implemented using three different hash functions, namely, SHA-1, SHA-512 (SHA-2), and SHA3-512 (SHA-3). The reasoning behind the chosen algorithms is in order to check whether the security of the hash functions itself influences the uniformity of the generated numbers. As for the translation function, this paper uses the simplest possible algorithm, namely using each digit in the generated hash string as is, and skipping the non-numeric characters within the hex string. Furthermore, in order to ensure that the uniformity of the generated numbers is measured correctly, each implementation with its own hash function would be run through 100000 iterations, or in other words, each function would generate 100000 hash strings' worth of random numbers.

For efficiency, this paper measures it by using the amount of time it takes for the algorithm to go through the 100000 iterations. As for the uniformity, this paper would measure it simply by-eye inspection of the histogram plot of the generated numbers.

V. Results

The result obtained from tests done using the method presented in the previous chapter would be presented and explored within this chapter.

TABLE I. Efficiency of Implementations With Different Hash Functions

Hash Function	Amount of Generated Numbers	Time Taken (s)	Numbers per Second
SHA-1	2,500,597	2.1602	1,157,576
SHA-512 (SHA-2)	7,999,295	6.4598	1,238,319
SHA3-512 (SHA-3)	7,997,897	6.5207	1,279,520
Randint (Python 3.6.5)	8,000,000	8.0385	995,210

First, the efficiency of the algorithms could be seen in Fig. 2, which compares the time taken to go through 100000 iterations and the amount of generated numbers between the implementations. As a baseline, the figure also shows the time it takes for the built-in *randint* function in Python 3.6.5. As can be seen, the amount of numbers generated by the algorithm implemented using the SHA-1 hash function is lower compared to the other implementations. The reason for this is that the length of a SHA-1 hash string is only 160 bits compared to the

512 bits using SHA-2 and SHA-3. But, the more important data to notice here is the amount of numbers generated per second by each implementation. As can be seen, every single algorithm generates more numbers per second compared to the built-in *randint* function from Python. This proves that the proposed algorithm is indeed efficient enough to be used for generating random numbers.

Next, the second important data to be explored here is the uniformity of the numbers generated by each implementation. From Fig. 3 below, it can be seen that for this specific translation function, the uniformity of every single implementation with different hash functions practically equals the uniformity of random numbers generated by a built-in Python function. This further proves the applicability of the algorithm to be used as a random number generator in real world scenarios.

VI. Conclusions and Further Research

As presented in Fig. 2 and Fig. 3, the result obtained from the implementations described in chapter IV appears to support the claims made in chapter I about the uniformity and efficiency of the proposed algorithm. With this, the algorithm could be further improved and explored in order to find out whether it is possible to use this algorithm as the random number generator in applications that require both efficiency and uniformity. This algorithm also provides at the very least a limited scalability in terms of memory and efficiency. This claim would have to be confirmed in large-scale applications that require very large numbers (in the hundreds of digits) as that would mean that the current implementation would require multiple iterations just to generate a single number.

That being said, there are several points that might be improved in further research on this topic. First of all, the translation function should be replaced by another function that would enable the generation of random numbers in a larger range of value. It is hoped that this new translation function would be able to preserve both the uniformity and the efficiency of the implementations that this paper explores. Secondly, future studies might explore the possibility that the efficiency and uniformity found in this paper is limited to few iterations. There is still a possibility that this algorithm would not scale for more significant applications. And finally, it should be checked whether there would ever be a practical point when the generated numbers would loop over due to hash collisions. Should such a

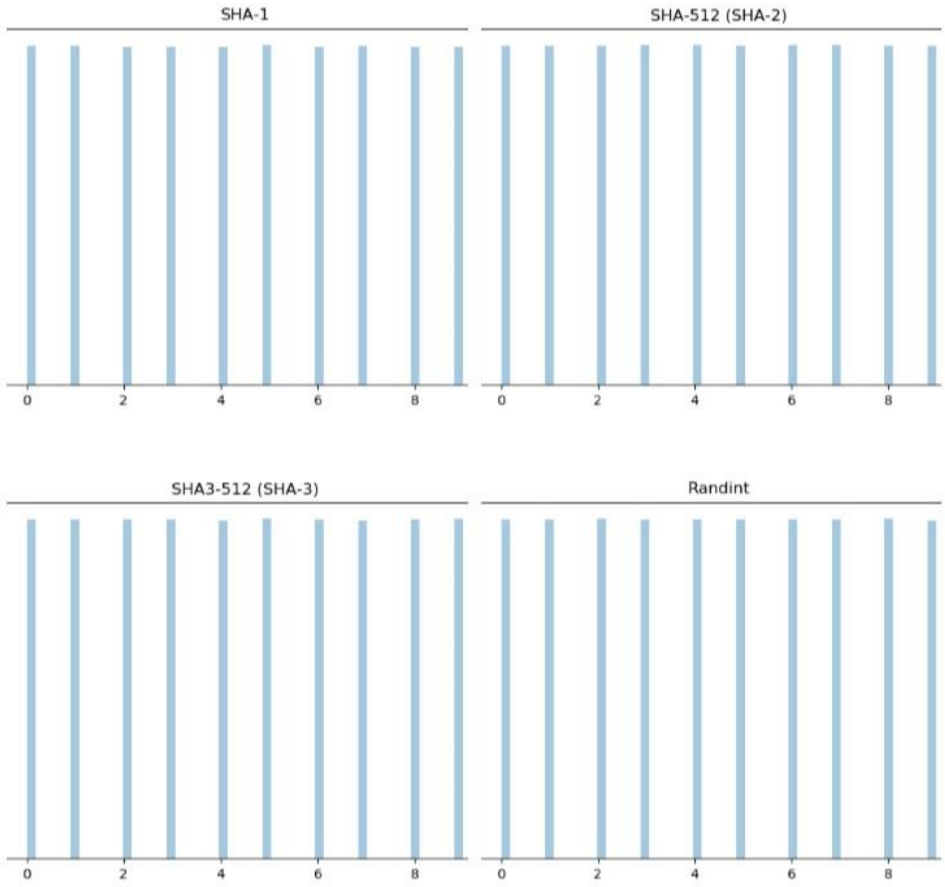

Fig. 3. Uniformity Comparison for Various Implementations

point be discovered, it would immediately render this algorithm useless for security-based applications.

ACKNOWLEDGEMENTS

In this section the author would like to thank Mr. Rinaldi Munir as the author's professor for teaching the knowledge that the author used during the writing of this paper. The author would also like to thank past researchers that have made their research available to the public as they presented the author with references that served as learning materials for the author.

REFERENCES

[1] V. Bagini and M. Bucci, "A Design of Reliable True Random Number Generator for Cryptographic Applications," Cryptographic Hardware and Embedded Systems Lecture Notes in Computer Science, pp. 204–218, 1999.

[2] "Hash," *Hash Definition*. [Online]. Available: https://techterms.com/definition/hash. [Accessed: 04-May-2019].

[3] *MD5 considered harmful today*. [Online]. Available: https://www.win.tue.nl/hashclash/rogue-ca/. [Accessed: 04-May-2019].

[4] Q. H. Dang, "Secure Hash Standard," 2015.

[5] S. Chen and C. Jin, "An Improved Collision Attack on MD5 Algorithm," *Information Security and Cryptology Lecture Notes in Computer Science*, pp. 343–357.