Report: Nino256 Hash Algorithm

Table of Contents

1. **Design and Architecture**
2. **Permutation Functions**
    1. **Permutation Function 1**
    2. **Permutation Function 2**
    3. **Permutation Function 3**
3. **Design of Permutation Functions**
4. **High-Level Mathematics**
5. **Testing results**
6. **Performance**
7. **Uses**

Design and Architecture

The Nino256 Hash algorithm is a 256-bit cryptographic algorithm that aims to generate a hash value from arbitrary length input data. The algorithm consists of three permutation functions, `permute_box1`, `permute_box2`, and `permute_box3`, and a hash function called `nino256_init`, `nino256_update`, `nino256_final`, and or convenience `nino256sum`.

The `nino256sum` function is the main entry point of the algorithm. It takes the following parameters: `buffer` (the input data), `length` (the length of the input data), and `digest` (the output hash value).

The function processes the input data in 32-byte chunks, loading the data into four state variables (`state_1`, `state_2`, `state_3`, and `state_4`). Then, the function performs the specified number of rounds for each chunk, repeatedly calling the three permutation functions on the state variables.

After processing all the data chunks, the resulting state variables are converted to a buffer and stored as a 256-bit hash value in the `digest` array.

Permutation Functions

The permutation functions, `permute_box1`, `permute_box2`, and `permute_box3`, transform the algorithm's internal state during each round. These functions operate on four 64-bit unsigned integers (`a`, `b`, `c`, and `d`), representing the algorithm's state.

The permutation functions use bitwise operations to achieve their non-linear transformations. The se bitwise operations operate on the state variables to increase the diffusion.

Permutation Function 1

$$\begin{gathered}d \leftarrow d \oplus (((a \ll 32) \& \sim (d \gg 32)) \oplus \sim a) \\ c \leftarrow c \oplus (((b \ll 48) \& \sim (c \gg 16)) \oplus \sim b) \\ b \leftarrow b \oplus (((c \ll 32) \& \sim (b \gg 32)) \oplus \sim c) \\ a \leftarrow a \oplus (((d \ll 16) \& \sim (a \gg 48)) \oplus \sim d)\end{gathered}$$

Permutation Function 2

$$\begin{gathered}a \leftarrow a \oplus (\mathrm{ROTRIGHT}((a \oplus b) \& \sim c, 20) \ll (\mathrm{ROTRIGHT}(d, 12) \mod 14)) \\ b \leftarrow b \oplus (\mathrm{ROTRIGHT}((b \oplus c) \& \sim d, 26) \ll (\mathrm{ROTRIGHT}(a, 17) \mod 14)) \\ c \leftarrow c \oplus (\mathrm{ROTRIGHT}((c \oplus d) \& \sim a, 15) \ll (\mathrm{ROTRIGHT}(b, 29) \mod 14)) \\ d \leftarrow d \oplus (\mathrm{ROTRIGHT}((d \oplus a) \& \sim b, 37) \ll (\mathrm{ROTRIGHT}(c, 47) \mod 14))\end{gathered}$$

Permutation Function 3

$$\begin{gathered}a \leftarrow a \oplus (\mathrm{ROTRIGHT}((a \,\&\, b) \oplus \sim c, d \mod 64) \gg (a \mod 16)) \\ b \leftarrow b \oplus (\mathrm{ROTRIGHT}((b \,\&\, c) \oplus \sim d, c \mod 63) \gg (b \mod 17)) \\ c \leftarrow c \oplus (\mathrm{ROTRIGHT}((c \,\&\, d) \oplus \sim a, b \mod 62) \gg (c \mod 18)) \\ d \leftarrow d \oplus (\mathrm{ROTRIGHT}((d \,\&\, a) \oplus \sim b, a \mod 61) \gg (d \mod 19))\end{gathered}$$

Design of Permutation Functions

The permutation functions in the Nino256 Hash algorithm are based entirely on bitwise operations and rotations. These operations are chosen to introduce diffusion, a fundamental property required in cryptographic algorithms.

The rotations shift the bits of the input variables, introducing non-linear transformations which help spread individual bits' influence throughout the state. The bitwise XOR operations combine the bits from different state variables to further increase the complexity and pseudo-randomness.

The additional mixing steps involve bitwise AND, NOT, and MODULO operations, introducing more intricate relationships between the state variables. These mixing operations ensure that the output of each permutation is dependent on all the input variables, making it difficult to analyze or predict the resulting state.

High-Level Mathematics

The Nino256 Hash algorithm relies on bitwise and discrete arithmetic operations on 64-bit unsigned integers. Therefore, the algorithm does not involve complex mathematical equations or advanced mathematical concepts. This low complexity yields a simple implementation and great potential for hardware acceleration.

The rotations, XOR operations, bitwise AND, and bitwise NOT operations used in the algorithm are basic operations that operate at the bit level, manipulating the individual bits of the state variables to achieve the desired non-linear transformations.

While the algorithm does not rely heavily on strict mathematical concepts, it leverages properties of bitwise operations and mixing to create a uniformly distributed cryptographic hash function.

Testing Results

The Nino256 Hash algorithm has demonstrated promising results regarding cryptographic strength during comprehensive automated testing. A substantial volume of testing data, amounting to 70 GB, was successfully generated utilizing a hash-based Cryptographically Secure Pseudorandom Number Generator (CSPRNG) constructed with 128 rounds of the Nino256 hash algorithm.

The generated CSPRNG data, including the Dieharder test suite, has undergone a rigorous automated assessment and has exhibited exceptional performance and security.

The construction employed for generating the data is outlined as follows:

```
// To generate 70 GB of PRNG data from the `hash_function`

void nino256sum(uint8_t *buffer, uint64_t length, uint8_t *hash);

int main()
{
        char csprng_state[32];
        FILE *output = fopen("csprng.bin", "wb");
        for (long iteration = 0; iteration < (75000000000 / 32);
iteration++)
        {
                nino256sum(csprng_state, 32, csprng_state);
                fwrite(csprng_state, 1, 32, output);
        }
        fclose(output);
}
```

These results showcase the robustness and reliability of the Nino256 Hash algorithm as a viable option for cryptographic applications requiring secure and high-quality pseudorandom number generation.

Performance

The Nino256 Hash algorithm has spectacular performance in a software-based implementation. The algorithm was implemented in C and compiled with `-O3` using the `gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)` compiler. The tests were performed on an `11th Gen Intel(R) Core(TM) i9-11950H @ 2.60GHz` CPU on bare metal (no Virtual Machine).

The test was performed on a 70 GB file generated by a CSPRNG to avoid CPU catching.

In this table, sha512sum and md5sum are the standard GNU Linux implementations.

Time Stat	Real	User	Sys	CPU Utilization
nino256sum	75.78s	44.27s	28.41s	95%
sha512sum	214.46s	148.43s	29.83s	83%
md5sum	148.18s	77.06s	33.71s	74%

In this table, sha256, sha512, and md5 are highly optimized by `OpenSSL 1.1.1f  31 Mar 2020`.

Algorithm	Real	User	Sys	CPU Utilization
nino256sum	91.96s	46.11s	34.06s	87%
SHA256	78.38s	40.17s	29.47s	88%
SHA512	126.18s	86.76s	30.37s	92%
MD5	115.01s	75.44s	30.53s	92%

Note that these other standard hash functions have assembly code, hardware, and complex algorithm optimizations to speed things up. Unfortunately, the nino256sum implementation used in these metrics did not have any of those improvements.

Uses

The Nino256 Hash algorithm can be used as a general-purpose cryptographic hash function. Hash functions are widely used in various applications, including data integrity verification, password storage, digital signatures, and message authentication codes.

The hash function produces a fixed-size (256-bit) output, regardless of the input size. This property makes it suitable for applications that require a constant-size representation of arbitrary data.

It is important to note that the security and suitability of the Nino256 Hash algorithm for specific use cases may require further analysis and evaluation. Therefore, it is recommended to consult with cryptography experts and undergo thorough security assessments before deploying custom cryptographic algorithms in real-world applications.