Optimizing Hash Strategy to Avoid Birthday Attack

Yiou Zhao
Beijing-Dublin International College at BJUT, Beijing University of Technology, Beijing, 100124, China
Corresponding E-mail: angela@cas-harbour.org

Abstract: It is necessary to optimize hash codes to prevent birthday attacks. First of all, it is necessary to know what hash codes are, what birthday attacks are, and how to attack them. In this article, there are two ways to avoid birthday attacks: enlarge hash space and selection of magic number. After the implementation of different code and different magic number selection, the most stable and secure one will be chosen.

1. Introduction
Hash codes represent a feature of an object, such as whether a certain two strings are equal or not, and if their hash codes are equal, the two strings are equal. Secondly, hash code is an algorithm of data structure. Common algorithms for hash codes are as follows. First, Hash Code of Object class. The structure of the memory address of the returned object is processed. Because the memory address of each object is different, the hash code is also different. Second, Hash Code of String class. According to the contents of strings contained in String class, a special algorithm is used to return hash codes. As long as the contents of strings are the same, the hash codes are the same. Third, Integer class, the hash code returned is the integer value contained in the Integer object, such as Integer I1 = new Integer (100), i1. Hash code value is 100. Thus, two Integer objects of the same size return the same hash code.

The rapid development of Internet technology promotes the development of human society. Information resources have made special contributions to the development of human society due to their universality, universality, value-added, processability and multi-effect. In order to ensure the integrity, reliability and availability of information resources, cryptography technology is needed, and Hash function is an important content in cryptography. The Hash function satisfies the unidirectional characteristic, that is, it can only get ciphertext from plaintext but not from ciphertext, and only the encryption process can not be decrypted. In addition, arbitrary length message inputs are hashed to output a fixed length Hash value. The unidirectional and fixed length characteristic of Hash function enables it to generate digital fingerprint (message digest, Hash value) of the message, so it is widely used in the fields of file verification, data integrity authentication, signature protocol, digital signature, digital encryption and so on.

At present, there are mainly MDx series and SHA series, including MD5, HAVAL, ripemd-12851, etc., and SHA series, including sha-0, sha-1, sha-256, etc. In the Hash algorithm, MD5 and sha-1 are the most widely used. In 2005, professor Wang Xiaoyun analyzed and improved the attack technology, and made collision attacks on sha-0 and sha-1 algorithms. These attacks pose a serious threat to the security of existing Hash functions and promote the development and research of new Hash algorithms. At present, the attack algorithms for Hash function can be divided into two general algorithms and specific algorithms. General algorithms include birthday attack, exhaustive attack,
midstream encounter attack and so on. Generally, this algorithm can attack all Hash algorithms. According to the current computing power of computers, the complexity of existing Hash algorithms meets the security requirements, and the computational complexity of attacks using general algorithms is generally large. However, with the rapid growth of computer operation speed and the popularity of quantum computing, the security of Hash algorithm will be greatly challenged. The specific algorithm makes use of the inherent structure defect of Hash function, and the attacker can only attack a certain Hash algorithm or a certain type of Hash algorithm, such as the differential attack of Wang Xiaoyun, the DOBBERTIN’s algebraic attack and so on. In the algebraic attack, the high bit in the MD5 operation process cannot be quickly and fully confused, and the attack is carried out by constructing two different 512-bit message groups and selecting the initial value to the semi-free initial collision. The difference attack is to attack the specific algorithm by finding effective difference and difference path when the high bit cannot be quickly and fully confused when using MD5 operation [1].

This article will focus on how to avoid birthday attacks.

2. What is birthday attack?
A hash is a mapping of different inputs into unique, fixed-length values (also known as "hashes"). It is one of the most common software algorithms. If different inputs get the same hash, there will be a "hash collision".

How many students does a class need to have to ensure that each student's birthday is different? The answer is surprising. If the probability of at least two students having the same birthday is less than 5%, then there can only be seven students in the class. In fact, there is a 50% chance that a class of 23 will have at least two classmates with the same birthday; Classes with 50 students have a 97% chance, while classes with 70 students have a 99.9% chance. This means that if the hash space is 365, there's a 50% chance of a collision if you compute 23 hash values. In other words, the probability of a hash collision is much higher than you might think. The number of computations required for a hash collision is on the order of magnitude of the square root of the value space. This kind of attack is called birthday attack when the hash space is not big enough to make the collision.

The most effective way to prevent hash collisions is to enlarge the hash space. There is a 1 in 65,536 chance of a collision between 16 binary hashes. In other words, if there are 65,537 users, there must be a collision. The length of the hash expands to 32 bits, and the probability of a collision drops to one in 4,294,967,296. Longer hashes mean more storage, more computation, and affect performance and cost. Developers must choose between security and cost [2].

3. Optimize the hash strategy
In a container that needs to be hashed, the most important thing is to avoid collisions. A collision is when two or more keys map to the same location. This also means that need to do some extra work to check if a particular key is the one you need, since there are now multiple keys in the same location. Ideally, each location should have at most one key.

| Mask a long | Use the mask on the result after string.hashCode () returns the value | Use the mask for hashmap.hash ( The result after string.hashCode () ) returns the value |
|-------------|-----------------------------------------------------------------------|--------------------------------------------------------------------------|
| 32          | There is no conflict                                                  | There is no conflict                                                      |
| 16          | 1 the conflict                                                       | 3 the conflict                                                            |

Table 1. The collisions of different keys
Took a Hash Map with a load factor of 0.7 (the default) and a range of 512. Some 30 percent of the collisions occur after a low 9-bit mask is used, even if the original data is unique.

```
static final int hash (Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

Figure 1. Result of the process.

This method mixes the high and low order of the original hash to reduce the randomness of the low order part. The high conflict situation in the above example can be alleviated by this means.

### 4. Selection of Magic Number

Magic Numbers are important. Sometimes the Numbers aren’t right. This is why a code structure is also needed to produce output in low worst-case scenarios, even if the magic number is incorrectly selected. Choosing the magic number does make a difference, but there are so many Numbers worth trying. You need to write a program that randomly selects enough cases to test [3].

| hash function       | Optimal multiplier | Minimum number of conflicts | Worst multiplier | Maximum number of conflicts |
|---------------------|--------------------|-----------------------------|-----------------|---------------------------|
| hash()              | 130795             | 81 collisions               | 126975          | 250 collisions            |
| xorShift16(hash())  | 2104137237         | 68 collisions               | -1207975937     | 237 collisions            |
| addShift16(hash())  | 805603055          | 68 collisions               | -1040130049     | 243 collisions            |
| xorShift16n9(hash())| 841248317          | 69 collisions               | 467648511       | 177 collisions            |

### 5. Key parts of the code

```
public static int hash(String s, int multiplier) {
    int h = 0;
    for (int i = 0; i < s.length(); i++) {
        h = multiplier * h + s.charAt(i);
    }
    return h;
}
```
Figure 2. Select magic numbers

It can be found that if provide a good multiplier, or a multiplier that just works for the key set, it makes sense to multiply each hash value by the sum of the next character. By contrast, only 81 collisions occurred when 130795 was used as multiplier for the key set tested, while 103 collisions occurred when 31 was used as multiplier [4].

If use the perturbation function at the same time, the conflict will be reduced to about 68 times. Such a collision rate is close to the effect of an array of buckets: don't take up more memory, but reduce the collision rate [5].

But what happens when you add a new key to the hash set? Can magic Numbers continue to work? It is with this in mind that worst-case collision rates are studied to determine which code structures perform better in the face of a wider range of input possibilities. The worst performance of hash() is 250 collisions: 70% of key collisions are really bad. The perturbation function improves the situation, but it's still not good enough. Note: if you choose to add the shifted value instead of separating, the result will be worse [6].

However, if choose two shifts - not just mixing the high and low parts, but mixing the four different parts of the hash value from the four parts of the hash function - will find that the worst-case collision rate decreases dramatically. So I think that if our structure is good enough and the effect of magic numbers is low enough, it will be less likely to get bad results when the selected key assembly changes.

6. What happens if choose to add instead of exclusive or in a hash function?

Using XOR instead of adding together in the perturbation function may get better results. $h = \text{multiplier} \times h + \text{s.charat}(i)$;

Replace to

$h = \text{multiplier} \times h \oplus \text{s.charat}(i)$;

Table 3. XOR in hash

| hash function | Optimal multiplier | Minimum number of conflicts | Worst multiplier | Maximum number of conflicts |
|---------------|--------------------|-----------------------------|------------------|----------------------------|
| hash()        | 1724087            | 78 collisions               | 247297           | 285 collisions              |
| xorShift16(hash()) | 701377257        | 68 collisions               | -369082367       | 271 collisions              |
addShift16(hash()) 1537823509 67 collisions -1409310719 290 collisions
xorShift16n9(hash()) 1638982843 68 collisions 1210040321 206 collisions

In the best case, the performance is slightly better, but in the worst case, the conflict rate is significantly worse. From this it can see that the importance of magic number selection has increased, that is to say, the selection of keys will have a greater impact. Considering that theselection of keys may change over time, this option seems dangerous [4].

7. Why Choose Odd Numbers as Multipliers?
When multiplied by an odd number, the position of the result may be either 0 or 1; because 0 * 1 = 0, 1 * 1 = 1. However, if multiplied by an even number, the lowest bit must be 0. That is to say, this bit no longer changes randomly. Let’s see what happens if repeat the previous test, but only use even numbers.

Table 4. Result of choose odd numbers

| hash function     | Optimal multiplier | Minimum number of conflicts | Worst multiplier  | Maximum number of conflicts |
|-------------------|--------------------|-----------------------------|-------------------|----------------------------|
| hash()            | 82598              | 81 collisions               | 290816            | 325 collisions             |
| xorShift16(hash())| 1294373564         | 68 collisions               | 1912651776        | 301 collisions             |
| addShift16(hash())| 448521724          | 69 collisions               | 872472576         | 306 collisions             |
| xorShift16n9(hash())| 1159351160        | 66 collisions               | 721551872         | 212 collisions             |

If lucky enough and the magic numbers are right, the results will be as good as those in odd numbers. But if bad luck happens, the result may be bad. 325 conflicts, that is to say, only 27 out of 512 barrels were used.

For the CITY, Murmur, XXHash it has the following characteristics: This strategy reads 64 data at a time, faster than reading bytes by bytes. The effective values obtained are two 64-bit long values. RMS reduced to 64 bits (As a result, more constant multipliers are used). The perturbation function is more complex.

Using long hash values in implementation because: It need optimize 64-bit processors and the longest data type in Java is 64-bit. If hash set is large (millions), it's hard to keep 32-bit hash values unique.

By studying the generation of hash values, the number of collisions between 352 keys was reduced from 103 to 68. At the same time, if the key set changes, it will also reduce the impact of the change. This does not require more memory or more computation time. Hash optimization can still choose to use more memory. By contrast, doubling the size of the bucket array can improve performance at best, but you still face an old problem: mismatches between magic Numbers and key sets can lead to serious conflicts.

8. Conclusion
When the key set is stable, the probability of collision can be significantly reduced by adjusting the hash strategy. You also have to test how bad things can get if you change the key set without
reoptimization. Combining these two approaches, you can develop a hash strategy that improves performance without requiring more memory.

Reference
[1] Bellare, M., Canetti, R., Krawczyk, H., et al. (1996) Keying Hash Functions for Message Authentication. In: international cryptology conference, Santa Barbara, CA, USA, 1-15.
[2] Venkatesan, R., Koon, S. M., Jakubowski, M. H., et al. (2000) Robust image hashing. In: IEEE international conference on image processing, Vancouver, BC, Canada, 664-666.
[3] Black, J., Halevi, S., Krawczyk, H., et al. (1999) UMAC: Fast and Secure Message Authentication. In: international cryptology conference, Santa Barbara, CA, USA, 216-233.
[4] Otsuka, T., Fujimoto, R., Utsuno, Y., et al. (2001) Magic Numbers in Exotic Nuclei and Spin-Isospin Properties of the NN Interaction. J. Physical Review Letters, 87(8).
[5] Bosman, E., Razavi, K., Bos, H., et al. (2016) Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In: IEEE symposium on security and privacy, San Jose, CA, USA, 987-1004.
[6] En, B. (2004) A One-Way Hashing Algorithm with Variable Length of Output. J. Computer Science.