DJB2 Hash

Dizon, Ken Shamrock L.

Simbahan, Patrick

Yang, Shin Yaw

Hashing is an algorithm that produces an output of garbled data through mathematical functions. These operations range from bit manipulation such as XOR and the four basic mathematical operations addition, subtraction, multiplication, and division. Each output produced by a hash function is of a constant length, which implies that regardless how long an input is, the output will have a uniform size. This type of algorithm is unidirectional (one-way) denoting data that has undergone data cannot be retrieved back to its initial state.

These algorithms have three (3) primary components: input, function, and hash value. The input is any data type that can be represented in bits. Although strings are the most commonly hashed data type, other data types, such as integers and floats can also be hashed (Sobti & Geetha, 2012). For example, the string “hello, world” can be used as an input and processed by the hashing function, which contains the set of rules that will be followed in generating the hash value. Each hash value is supposed to be unique for every word. For instance, the strings “hello world” and “helloworld” should produce distinct hash values. However, from time to time two different inputs produce the same hash value, which is called a collision.

As part of the authors’ Computer Architecture class, they were tasked to create a hash function with the MIPS assembly language. MIPS is a low level language that was created by MIPS Technologies to be used for MIPS architecture processors. Since modern day computers run on the x86 architecture, the hash function that they made was written with the MIPS Assembler and Runtime Simulator (MARS). Thus, it is necessary to have MARS and Java to be able to run this program.

Since the implementation of cryptographic hash functions (CHF) is laborious, the authors opted to construct a non-cryptographic hash function (NCHF) instead. Compound this with the limitation of programming in the MIPS architecture which only features a limited amount of memory, processing power, and instruction sets. While not as secure as CHFs, there is a wide spread use of NCHFs in the field of computer science as these make it possible to search information in a set quickly within a set time of *O(1)*. As a result, NCHFs are used in lexical analysis, databases, compilers, and networking (Estébanez, et al. (2014).

The DJB2 hash algorithm was chosen by the authors as it is a relatively simple and robust hash function. It was created by computer scientist Daniel J. Bernstein, hence the name. Developers are fond of this function due to its simplicity and ease of modification. In addition to these factors, a research conducted by Kuznetsov et al. (2021), concluded that djb2 was one of the fastest hash algorithms. For instance, the command and control framework, Havoc, which has been found to be able to bypass Windows 11 Defender, uses a DJB2 algorithm for hashing which C5pider retrieved from the vxunderground community account from GitHub (C5pider, 2022).

In terms of hashing performance, the DJB2 algorithm does fairly well compared to other NCHFs. Stack Exchange user Ian Boyd rigorously tested NCHFs which included DJB2. His findings strongly suggest that this hash algorithm's rate of collision is rare as well as having a good distribution. Additionally, he included the word pairs that produced the same hash values (Boyd, 2012).

Despite these strengths, it is important to understand the weaknesses of this algorithm. As it only produces hash values within 32 bit unsigned integers, the range of values is severely limited which limits the amount of values it can uniquely produce. Moreover, it relies on bitshift and addition operation only which further inhibits its performance. This results in a poorer distribution of values making this algorithm not suited for actual security purposes.

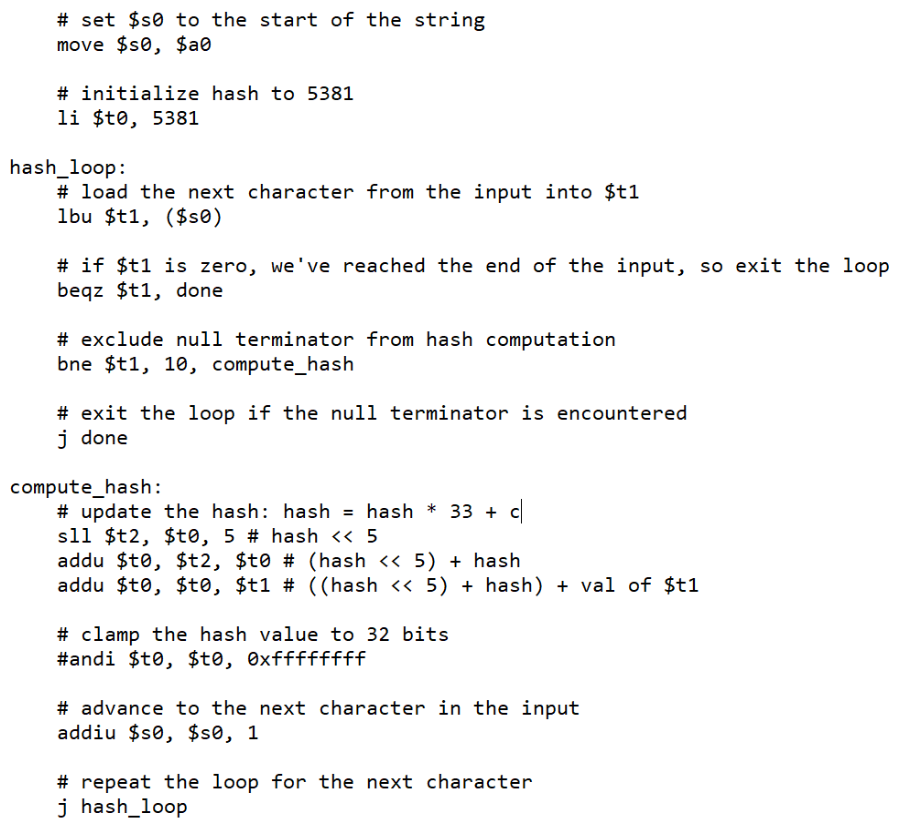
Overall, DJB2 hash function is a well performing algorithm that offers both great speed and decent collision rate of hash values albeit with some drawbacks which makes it a good candidate for this project. Furthermore, its simple structure allows for easy modifications to be made in order to suit the needs of a project making it an excellent candidate for the limited programming environment of the MIPS architecture.

DJB2 hash algorithm works with the following steps:

1. Initialize the hash with 5831. This was found by Bernstein to produce the best distribution of hash values.
2. Iterate through each character in the input string: For each character in the input string, the algorithm performs the following steps:
    a. Shift the hash value to the left by 5 bits. This is equivalent to multiplying the value by 33.
    b. Combine the hash value before shift and hash value after shift and set it as the current hash value.
    c. Add the integer representing the character to the current hash value.
3. Return the final hash value: Once all characters have been iterated through, the final hash value is returned.

MIPS implementation

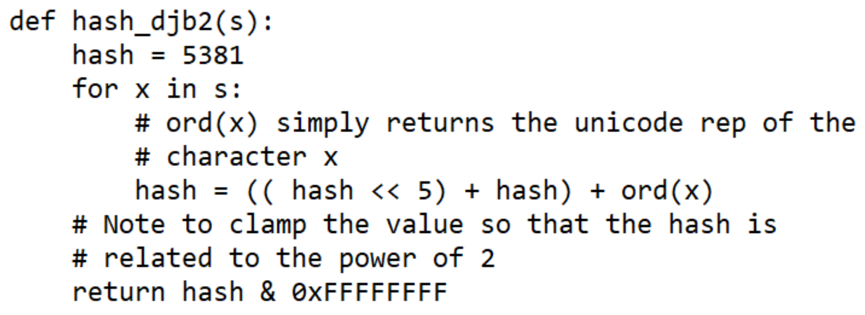
1. The following registers are used by the program:
    1.1. $t0 - register where the initial hash value 5381 is stored
    1.2. $t1 - pointer used to iterate over each character of the string
    1.3. $t2 - temporarily holds $t0 that has been left shifted by 5
    1.4. $s0 - holds the string inputted by user
2. The program then iterates over each character of the string
    2.1. First, shift the bits of the hash held by $t0 and store it to $t2
    2.2. Add the value of $t2 to the $t0 and store it to $t0
    2.3. After add $t0 and $t1 then store sum at $t0
    2.4. Increment $s0 and repeat steps until null pointer is met



```
    # set $s0 to the start of the string
    move $s0, $a0

    # initialize hash to 5381
    li $t0, 5381

hash_loop:
    # load the next character from the input into $t1
    lbu $t1, ($s0)

    # if $t1 is zero, we've reached the end of the input, so exit the loop
    beqz $t1, done

    # exclude null terminator from hash computation
    bne $t1, 10, compute_hash

    # exit the loop if the null terminator is encountered
    j done

compute_hash:
    # update the hash: hash = hash * 33 + c
    sll $t2, $t0, 5 # hash << 5
    addu $t0, $t2, $t0 # (hash << 5) + hash
    addu $t0, $t0, $t1 # ((hash << 5) + hash) + val of $t1

    # clamp the hash value to 32 bits
    #andi $t0, $t0, 0xffffffff

    # advance to the next character in the input
    addiu $s0, $s0, 1

    # repeat the loop for the next character
    j hash_loop
```

DJB2 MIPS code snippet.

The output of this implementation produces the same output of the Python DJB2 implementation made by Mari Batilando (2022).



```
def hash_djb2(s):
    hash = 5381
    for x in s:
        # ord(x) simply returns the unicode rep of the
        # character x
        hash = (( hash << 5) + hash) + ord(x)
    # Note to clamp the value so that the hash is
    # related to the power of 2
    return hash & 0xFFFFFFFF
```

DJB2 Python code snippet.

In addition to generating the same hash values as the python implementation, the DJB2 MIPS function made by the authors also had the same collisions as the one's Boyd produced, further validating the correctness of the hash function. The word pairs and there hash values are:

hetairas	mentioner	3940087062
heliotropes	neurospora	874715923
depravement	serafins	218169280
stylist	subgenera	2945051873
joyful	synaphea	121541278
redescribed	urites	555028097
dram	vivency	2090194761

Table containing words with colliding hash values.

As part of the final project requirements, the authors also had to combine the DJB2 hash algorithm made with a password validation program in MIPS. The following are the password requirements that must be followed for a valid input:

1. Password must be 8-16 characters long
2. Must contain at least one lowercase character
3. Must contain at least one uppercase character
4. Must contain at least one special character
5. Should have no white spaces (Including horizontal tab)

Validating the password was done by iterating over every character and checking where its ascii values fall within a range. If a whitespace or tab is detected, the program returns an invalid input and asks the user to input a password again. Otherwise, each character will be checked, below are the range of values that determine what a character represents:

1. 65 - 90 = uppercase char
2. 97 - 122 = lowercase char
3. 48 - 57 = integer char
4. else = special char

References

Batilando, M. (2022, October 28). The DJB2 Hash Function. HackMD. https://hackmd.io/@sIQnCbQ0T56A3KLAiNrlhQ/r1GtXjBVP

Boyd, I. (2011, Feb 14). MurmurHash3: Is it good enough for passwords? [Response to the question "Which hashing algorithm is best for uniqueness and speed?"]. In Stack Exchange. https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed/49563#49563

C5pider. (2022, October 13). C5pider on Twitter: “Didn't modify anything. Used dbj2 from @vxunderground” [Tweet]. Twitter. https://twitter.com/C5pider/status/1580361349971857409

Embee Research (2022, April 5). Embee Research on Twitter: “3/ The hashing algorithm used is possibly a modification of DJB2. The usage of 0x1505/5381 gives this away. If not DJB2, the algorithm is of a highly similar structure.” [Tweet]. Twitter. https://twitter.com/embee_research/status/1579668726252974080

Estébanez, C., Saez, Y., Recio, G., & Isasi, P. (2013). *Performance of the most common non-cryptographic hash functions. Software: Practice and Experience, 44(6), 681–698.* doi:10.1002/spe.2179

Kuznetsov, A., Tymchenko, V., Kolhatin, A., & Shekhanin, K. (2021). Cryptographic Hash Functions Suitable for Use in Blockchain

Sobti, R., & Geetha, G. (2012). Cryptographic Hash Functions: A Review. IJCSI International Journal of Computer Science Issues, 9(2).

vxunderground. (n.d.). vxunderground. GitHub. https://github.com/vxunderground