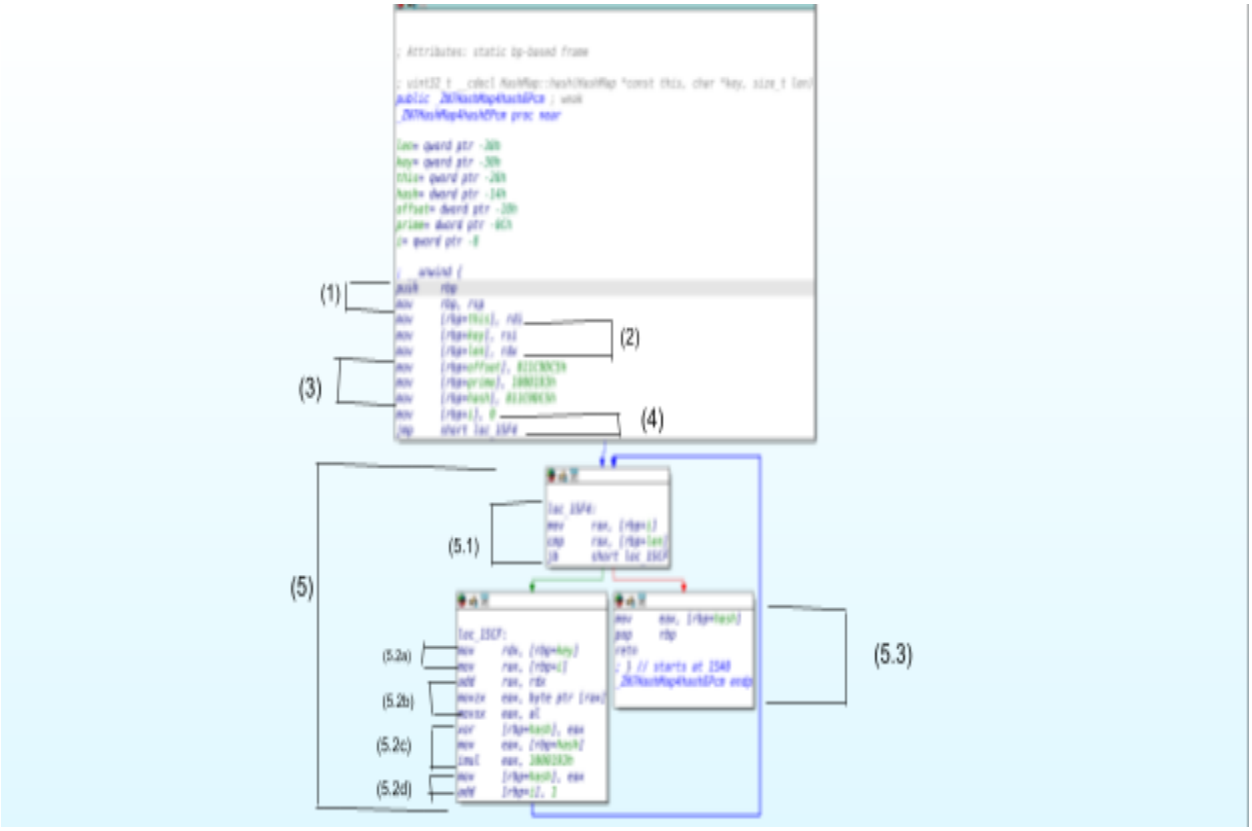

This hash function uses the FNV-1a hashing algorithm.

FNV-1a hash [edit]

The FNV-1a hash differs from the FNV-1 hash only by the order in which the multiply and XOR is performed:[9][11]

```
algorithm fnv-1a is
    hash := FNV_offset_basis

    for each byte_of_data to be hashed do
        hash := hash XOR byte_of_data
        hash := hash × FNV_prime

    return hash
```

The above pseudocode has the same assumptions that were noted for the FNV-1 pseudocode. The change in order leads to slightly better avalanche characteristics.[9][12]

1. Function Preamble
2. Load in function arguments and class pointer into stack frame to use in loop
3. Load in the recommended 32 bit hex values as per the FNV guidelines. Note that the hash local variable is the same as the offset as hash var is set equal to offset.
4. Load in loop incrementer into stack frame and start the loop
5. Primary Loop
    - 5.1
        - ➢ Move current loop incrementer value to rax and compare to the length of the hashmap key. Interesting to note is that it uses jb rather than a jle.

Research shows that jb is unsigned and my implementation returns an unsigned 32 bit integer. Continue the loop if i is below the length of the key (5.2) or go to the function epilogue (5.3) otherwise

- 5.2 Hashing Loop
    a. Load in the key and loop incrementor for hashing
    b. Static casting. The program grabs the element to be casted with the add operation of the key and the loop index, then zero and sign extends that value to convert the signed int to a uint_32 (or unsigned int).
    c. The actual hashing steps with xor and a signed multiply using the recommended 32bit FNV_prime value
    d. Stores the updated hash value on the stack and increments the index for the loop
- 5.3
    ➢ Function Epilogue, returns the final hash value in eax