Numhash: a provably collision-resistant hash function based on the hardness of numerical k-dimensional matching

made by DEDMANWALKING
version 1.0 of this document
github.com/dedman24/numhash0

Abstract. Most cryptographic hash functions with provable security reductions are either too inefficient to be used practically or they do not fully utilise the data parallelism modern processors offer.
We introduce **NUMHASH**, a provably collision-resistant hash function that only uses addition and swaps in its compression function in a way that can be trivially parallelized in modern processors.

1 Notation

Vectors and Matrices. Vectors are denoted by one or multiple upper-case letters (e.g. A) and matrices are denoted by an upper-case boldface letter (e.g. **M**). *All vector and matrix entries are indexed from zero.* A vector's ith entry is denoted by the same letter, but lower-case and with the subscript i (e.g. x_i is the ith entry of the vector X). Similarly, the ith column of a matrix is denoted by an upper-case non-boldface letter with the subscript i (e.g. A_i is the ith entry of some matrix). A vector V_i's jth entry is denoted by the same letter, but lower-case and with the subscript i followed by a comma and by the subscript j (e.g. $v_{i,j}$ is the jth entry of the vector V_i).
Bitwise operations. & is used to denote the bitwise-and operation, || is used to denote string concatentation and <<< is used to denote bitwise left rotation. All additions are modulo 2^{64}.
Byte order. All integers are stored as their binary little-endian representation.
Interval notation. $x \in [a, b]$ is used to denote "$a \leq x \leq b$". $x \in [a, b)$ is used to denote " $a \leq x < b$". Thorought the document, when an integer is said to "lay within an interval" the above notation is implied (e.g. if x is said to be within the interval [a, b], it should be interpreted as $x \in [a, b]$). [1]

2 Specification

The compression function. Let C(B, **M**) be the numhash compression function, taking as input a vector B of 128 integers, a random matrix **M** and returning a 64-byte digest.
B is the bit string to be hashed; each integer within B is within the interval [0, 255]. **M** is a 128 × 8 matrix containing only integers within the range [0, 2^{64}); **M** is generated ahead-of-time and it stays constant thorought all calls to C. Any random matrix is fine, but the one used by numhash0 was generated by calling blake2xb on a zero-length empty string with an output length of 8192 bytes.
Let Scramble(M_i, h) be a function that shuffles an input vector holding 8 entries, each within the range [0, 2^{64}) ($m_{i,0}$ through $m_{i,7}$), based on h, an integer within [0, 255]. M_i is the ith vector of **M**.
Scramble swaps $m_{i,0}$ with $m_{i,1}$ if h & 2^0 is **TRUE**; $m_{i,1}$ is swapped with $m_{i,2}$ if h & 2^1 is **TRUE**; this is repeated for $m_{i,2}$ through $m_{i,7}$ with the exception that $m_{i,7}$ is conditionally swapped with $m_{i,0}$. The resulting scrambled vector is added to a vector OUT, which holds 8 entries within the interval [0, 2^{64}).
Scramble is called four times per $b_i \in B$, for a total of 512 calls. Each time, b_i is rotated left by two. On a binary processor, b_i is made out of eight bits; changing one bit within the input causes the output of Scramble to change by minumum 2 entries; calling Scramble multiple times and summing the result to OUT while rotating b_i by 2 causes OUT to change by min(2n, 8) entries, where n is the number of times Scramble is called per bit. It is obvious that, if one bit in B is changed, all of the output changes too.

The compression function is described through pseudocode as:

```
for $i$ = 0;  $i$ < 128; $i$++ do:
        OUT  = OUT + Scramble($M_i$, $b_i$);
        OUT  = OUT + Scramble($M_i$, $b_i$ <<< 2);
        OUT  = OUT + Scramble($M_i$, $b_i$ <<< 4);
        OUT  = OUT + Scramble($M_i$, $b_i$ <<< 6);
```

where OUT + Scramble(…) is an addition between the vector OUT and the vector returned from Scramble(…). M_i being reused should not affect the security of the hash function.

Security proof. The compression function can be proved to be collision-resistant assuming the Numerical 3-Dimensional Matching problem is not solvable in polynomial time on the instances provided by numhash. Finding a collision is reducible to the *Different Bound Numerical 512-Dimensional Matching* problem; that is, given 512 sets X_0 through X_{511}, each with 8 elements, find the subset of S = {$X_0 \times X_1 \times \ldots \times X_{511}$} so that every integer in S occurs exactly once and for every 512-uple {$x_{0,i}$, $x_{1,i}$, … , $x_{511,i}$} the property ($x_{0,i} + x_{1,i} + \ldots + x_{511,i} = out_i$) holds, where OUT is the output vector. We can map the *Numerical 512-Dimensional Matching* problem onto this by having all the entries within OUT be equal; the *Numerical 3-Dimensional Matching* problem can be mapped onto this problem by having the sets X_3 through X_{511} hold only zeroes.

All of these transformations can be done in polynomial time. Finding a collision for the numhash compression function is as hard as solving the *Numerical 3-Dimensional Matching* problem, which was proven to be NP-hard in [2].

I could not find any algorithm for Numerical k-Dimensional Matching faster than brute-forcing. This is impractical: it is as hard as finding a collision or a preimage on an ideal universal hash function.

Flaws within the compression function. C(msg) is used instead of C(msg, **M**) in this section. For all intents and purposes, the two are equal.

The output of C is *easily distinguishable* from a random oracle when not truncated: if you sum each 8-byte segment in OUT, the result is always the same regardless of the input (e.g. $out_0 + out_1 + out_2 + out_3 + out_4 + out_5 + out_6 + out_7$ = constant). In numhash0, this constant is always *0xc95ca4481e76c4dc*. As hinted above, this "constant sum" property does not hold for when the output is truncated, but it does not mean truncated numhash is indistinguishable from random data.

Numhash should never be used as a random oracle as of right now.

If part of the input to C is known, then the digest with that input modified can be trivially computed, even when the whole input is not known. We will call this the *"known-input modification attack"*.

This allows an attacker that knows C(message || secret) to compute C(message'|| secret) as long as they also know message and message' is equal in length to it, *even if they do not know secret*.

Adding a final compression function call makes this attack as hard as finding the preimage to C; therefore, *numhash may be used as a MAC*. This added compression function call slows down calls by up to 33%. Numhash0m, which is unkeyed, does NOT have this added compression function call, and numhash0km only has it when the length of the key is larger than 0.

Length extension attacks are not an issue thanks to the finalization call and the inclusion of the message length within the IV (see next section).

3 Hash construction

Let D(B, P) be C(P || C(B, **M**), **M**), where P is a vector with 8 integers within the interval [0, 2^{64}) holding the output from the previous iteration and B is the block to be hashed.

When first calling numhash0, P is initialized as the triple {outlen, inlen, keylen} padded with zeroes.

Outlen is the output length in bytes, inlen is the input length in bytes, keylen is the key length in bytes.
Let numhash0m(IN, inlen, outlen) be an hash function that uses D inside the Merkle-Damgård hash construction. IN (the input message vector) is divided into blocks of 128 bytes; the last block may be shorter, in which case it gets padded with zeroes.
In C code, the core part of numhash0 is:

```
// let BLOCK be a constant equal to 128, let NUMHASH_ROUND be a pseudonym of C
// let the bottom 8 entries of P be the vector described earlier with the same name

        uint64_t input_processed = 0;
        while(inlen > BLOCK){
                NUMHASH_ROUND(P + 8, (uint8_t*)in + input_processed);
                NUMHASH_ROUND(P, (void*)temp);
                inlen -= BLOCK;
                input_processed += BLOCK;
        }
        uint8_t buf[BLOCK];
        memcpy(buf, (uint8_t*)in + input_processed, inlen);
        memset(buf + inlen, 0, BLOCK-inlen);
        NUMHASH_ROUND(P + 8, buf);
        NUMHASH_ROUND(P, (void*)temp);
```

Numhash0km(IN, inlen, outlen, KEY, keylen) divides and pads the key like numhash0m does the message and it returns C(numhash0m(padded message || padded key, inlen, outlen)) if keylen > 0 or numhash0m(padded message, inlen, outlen) if keylen = 0. It sets the keylen parameter inside the P vector to to the length of the unpadded key.

4 Performance

In this section, numhash0 will be used to refer both to numhash0m and numhash0km.
All of the vector addition within the compression function can be performed in parallel via the AVX512 operation VPADDQ. Scramble is implemented via a for loop, but it could be done in hardware trivially if specialized instructions existed.
Since the matrix needed for numhash0 is 8KB large, caching plays a significant role. Two tests were performed; one that calls numhash0 directly and one that calls it 2^{16} times beforehand (each time with random data as the message) to ensure that **M** is in L1 cache. 16 readings were taken per test and their bandwidth was averaged. Both the speed of the executable produced by gcc and by clang will be compared.
UNCACHED-GCC means that the hash function was called directly (without allowing the cpu to cache data) and the program was compiled with GCC; CACHED-CLANG means that, within the program, numhash was called 2^{16} times beforehand (to allow it to cache **M**) and it was compiled with CLANG.
Everything was tested on an Intel Core i7-1165G7 laptop running MX linux while powered.
Different input sizes did not seem to have a significant effect on throughput for both cached and uncached tests for numhash0m. Numhash0km's speed on inputs of different sizes follows the formula:
$(throughput\_numhash0m*\lceil inlen/128\rceil)/(\lceil inlen/128\rceil + 1)$
Where throughput_numhash0m is how many bytes per second numhash0m can hash, shown in figure 1 below.

The results we found were:
(UNCACHED-GCC) 3.6MB/s on 1024-byte blocks
(UNCACHED-CLANG) 9.44MB/s
(CACHED-GCC) 12MB/s
(CACHED-CLANG) 38.53MB/s

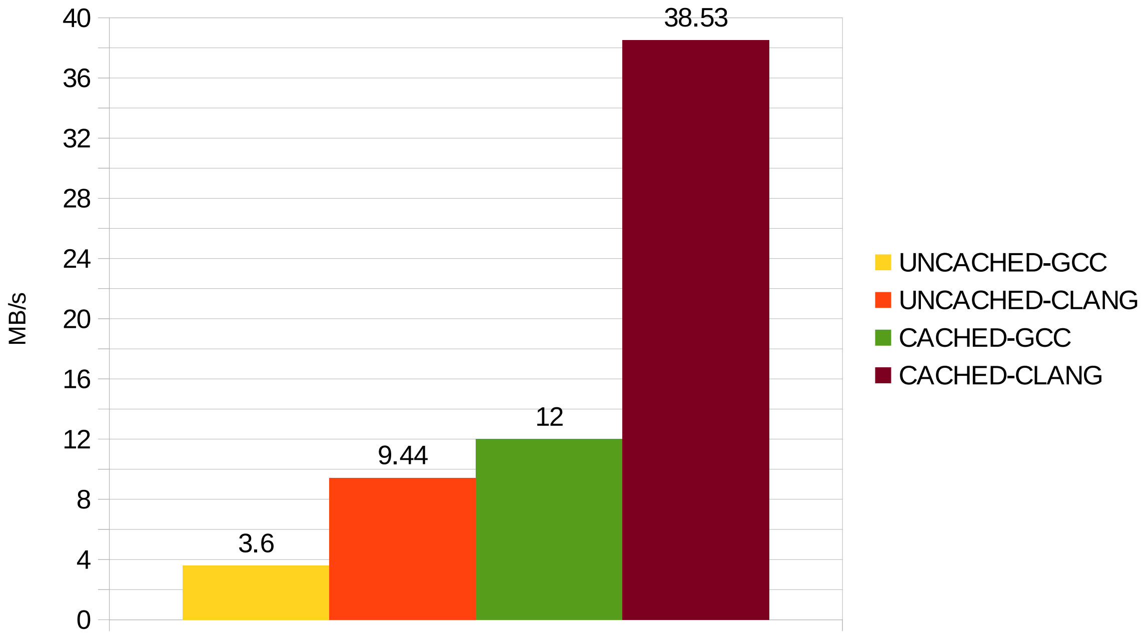

Figure 1: throughput of numhash0m when measured from our laptop

The speed difference between clang and gcc is so large because, while both unroll the loop, clang uses the CMOV instruction to perform swaps, while gcc uses branches, which are slower since branch prediction would have to predict what the input is.
It is impressive, however, that using predicated operations incurs such a large speedup; up to ~3.2x.
I found it frustrating trying to force GCC to generate conditional moves like CLANG did. No matter what I tried it refused to do so, even when the speedup would be significant.

5 References

[1] github.com/algorand/go-sumhash/blob/master/spec/sumhash-spec.pdf was were the wording for the initial “Notation” section was taken from.
[2] Caracciolo, Sergio; Fichera, Davide; Sportiello, Andrea (2006-04-28), One-in-Two-Matching Problem is NP-complete, arxiv.org/abs/cs/0604113