Benchmarking Zero-Knowledge Proofs with isekai *

Guillaume Drevon, Aleksander Kampa

December 2019

Abstract

We present the results of a benchmarking exercise for five ZK proof systems supported by the isekai verifiable computation framework. Because identical arithmetical circuits are used, a direct comparison becomes possible. Results are provided for three types of computations: dynamic memory access, array sorting and sha256 hashing.

Keywords: zero-knowledge, benchmarking.

1 Introduction

The recently released version 1.0 of the isekai verifiable computation framework supports the following proof systems: libsnark (Groth16 and BCTV14a), dalek (Bulletproof) and libiop (Aurora and Ligero). It is therefore now possible to directly compare several zero-knowledge proof systems, using identical arithmetic circuits. This paper describes the result of a benchmarking exercise of these proof systems for three types of computations: dynamic memory access, array sorting and sha256 computations.

This paper is structured as follows. We first provide an overview of isekai in Section 2, then describe the computations and the benchmarking setup in Sections 3 and 4. Section 5 focuses on proof times, Section 6 on proof sizes and Section 7 on verification times. After a brief conclusion, tables of detailed benchmarking results as well as the source code of C/C++ programs used are provided as appendices.

*Research supported by Fantom Foundation


citation: 

**2 About isekai**

Isekai [ise] is an open-source verifiable computation framework developed by Sikoba Research [Sik] with the support of Fantom Foundation [Fan] and written in Crystal language [Cry]. The isekai project was started in October 2018. The goal was to create a tool that would allow a non-specialist to use zero-knowledge proofs by connecting source code written in standard programming languages with zero-knowledge proof systems.

With the release of isekai 1.0 in November 2019, these goals have arguably been achieved to a reasonable degree:

1 — Thanks to its LLVM [LLV] frontend, isekai now works with C and C++ programs and potentially with any language with an LLVM frontend. There are still some limitations, of course: for example, only integers-like types are supported. Overall, however, it supports more language features than other projects in the ZK space.

2 — isekai is able to interface with several zero-knowledge proof libraries, with each having different properties such as no trusted setup, quantum resistance, compactness, etc. Specifically, isekai supports: libsnark (Groth16 and BCTV14a), dalek (Bulletproof) and libiop (Aurora and Ligero).

**3 Computations**

The first computation is simply doing dynamic memory access (think of a[b[i]]). Internally, isekai uses many conditionals (ifs) to handle this, and the number of conditions grows linearly with the array size. The tests are named cond10, cond100 and cond1000, with array sizes 10, 100 and 1000 respectively. Cond1000 has more than 6 million constraints. This shows the limitation of the current implementation, and although it can be slightly improved (we can easily reduce the constraints number from 3n to 2n), the best way to handle this is by implementing TinyRAM techniques. But that’s a topic for isekai 2.0!

The second computation is simply sorting an array and returning the median. The sorting involves many comparisons that are not ZKP friendly. As a result, we could not perform the test of sorting an array of 1,000 elements because it generated more than 5 million constraints. This once again shows the limitation of the current implementation. Sorting is best handled with dedicated constraints such as a dedicated ZKP_SORT function, but this would deviate from the isekai approach, which is to use regular programming language features. The tests are named med10 and med100 with array sizes of 10 and 100 respectively.

Finally, for the third test, we have selected a widely-used function: a sha256 computation. While the first two tests show isekai’s limitations, this one shows how powerful isekai can be. To implement a zero-knowledge proof of a sha256 computation, we simply took the first C++ implementation we found on the web and modified it slightly to make it compatible with isekai — the changes were easy and straightforward. Then, using isekai, we were immediately able to produce a proof of a hash computation using several proof schemes. The tests are named h32, h128, h512 and h1024 and compute the sha256 of a byte array of size 32, 128, 512 and 1024, respectively. h1024 generates around 1 million constraints.
4 Benchmarking setup

We have compared all the ZK schemes supported by isekai using identical arithmetic circuits. This gives consistent results, although there may still be some implementation issues, compiling options or system settings (such as curve choice) that can affect the results.

The computer used was a Lenovo T580 laptop with an i7–8550U processor (base frequency @ 1.80 GHz, turbo @ 4.00 GHz), 32 GB RAM and a 1 TB SSD hard drive. Note that for computations involving the most constraints, the entire memory was used and the system had to swap, which obviously affected performance.

The results are plotted against the number of constraints.

5 Proof Time

We start by looking at proof times. We see that Ligero has a very good performance, comparable to those of zk-SNARKs even without a trusted setup. However, it did not work on the last two tests, those with many constraints, for which we kept getting an ‘out of memory’ error. This may be due to the implementation, because we had to implement some padding regarding the number of variables and this has an impact on both performance and stability.
The graph does not always show an increase, meaning that the number of constraints is not the only factor affecting performance. The complexity of the constraints is also important.

6 Results: Proof and Trusted Setup Sizes

Next, we look at proof and trusted setup sizes. Bulletproof has the lowest performance overall but its proof size stays really low, considering that it does not require a trusted setup. This leaves room for a trade-off between proof size and proof time.

Plot 2: Proof Size vs. Trusted Setup

The proof size of zk-SNARKs is extremely small and constant: 209 bytes for the Groth16 scheme and 437 bytes for BCTV14a. However, the size of the trusted setup grows rapidly, up to 12 gb for 6 million constraints. Note that this is the raw data, and it turns out that this data compresses very well. One can expect a 10:1 ratio when compressing a trusted setup.

Note that Ligero’s performance comes at a cost: its proof size is much higher than that of Bulletproof (4 to 6 kb) and Aurora (200 to 600kb).
Plot 3: Size of Trusted Setup

Constraints

Bytes

10^3 10^4 10^5 10^6 10^7

-2 -1 0 1 2 3

Plot 4: Verification Time

Constraints

Seconds

10^{-2} 10^{-1} 10^0 10^1 10^2 10^3
7 Results: Verification Time

Finally, we look at the verification time, which is shown in Plot 4. It follows the same pattern as the proving time but is always significantly faster. As noted before, dynamic memory access is currently not handled efficiently in isekai, which results in slow verification times for the cond1000 computation.

8 Conclusion

In conclusion, there is no overall winner: all proof systems have different pros and cons, different properties and different security assumptions. As long as issues related to trusted setup size and generation are handled, zk-SNARKs clearly have the best performance. Bulletproofs manage an impressively small proof size without trusted setup, but Aurora has a faster proof time and is quantum resistant.

We note that new proof systems have shown up recently, for instance Fractal, an improvement over Aurora, or Marlin and Plonk, based on polynomial commitments. We hope to be able to test them in a future isekai benchmarking exercise.

References


Appendix 1: Benchmarking Data

The detailed results of the benchmarking exercise are provided in the tables below.

**Table 1:** Circuit and R1CS

<table>
<thead>
<tr>
<th>Name</th>
<th>Circuit*</th>
<th>R1CS*</th>
<th>Constraints</th>
<th>Witnesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond10</td>
<td>0.2</td>
<td>0.4</td>
<td>1,767</td>
<td>1,745</td>
</tr>
<tr>
<td>cond100</td>
<td>6.7</td>
<td>40.8</td>
<td>85,745</td>
<td>85,461</td>
</tr>
<tr>
<td>cond1000</td>
<td>600.0</td>
<td>4,181.0</td>
<td>6,266,234</td>
<td>6,263,354</td>
</tr>
</tbody>
</table>

* size in MB

**Table 2:** Proof time (seconds)

<table>
<thead>
<tr>
<th>Name</th>
<th>bc1tv4a</th>
<th>groth16</th>
<th>bulletproof</th>
<th>ligero</th>
<th>aurora</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic memory access</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond10</td>
<td>0.2</td>
<td>0.3</td>
<td>1.0</td>
<td>0.7</td>
<td>0.5</td>
</tr>
<tr>
<td>cond100</td>
<td>12.3</td>
<td>10.3</td>
<td>38.6</td>
<td>12.7</td>
<td>30.6</td>
</tr>
<tr>
<td>cond1000</td>
<td>1,378.2</td>
<td>1,277.0</td>
<td>3450.0</td>
<td>-</td>
<td>2,918.5</td>
</tr>
</tbody>
</table>

| Sorting an array |
| med10  | 0.6     | 0.5     | 2.3         | 0.7    | 1.3    |
| med100 | 77.4    | 63.8    | 283.5       | 27.2   | 108.5  |

| sha256 computation |
| h32    | 5.7     | 4.7     | 21.2        | 4.4    | 13.0   |
| h128   | 28.8    | 24.2    | 82.7        | 23.6   | 62.3   |
| h512   | 129.6   | 113.5   | 307.5       | 122.9  | 171.0  |
| h1024  | 335.0   | 328.7   | 717.8       | -      | 428.3  |
### Table 3: Verification time (seconds)

<table>
<thead>
<tr>
<th>Name</th>
<th>bctv14a</th>
<th>groth16</th>
<th>bulletproof</th>
<th>ligero</th>
<th>aurora</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dynamic memory access</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond10</td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>cond100</td>
<td>2.6</td>
<td>2.2</td>
<td>9.6</td>
<td>6.4</td>
<td>3.7</td>
</tr>
<tr>
<td>cond1000</td>
<td>416.0</td>
<td>374.5</td>
<td>887.0</td>
<td>-</td>
<td>506.6</td>
</tr>
<tr>
<td><strong>Sorting an array</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>med10</td>
<td>0.6</td>
<td>0.0</td>
<td>0.4</td>
<td>0.6</td>
<td>0.1</td>
</tr>
<tr>
<td>med100</td>
<td>9.3</td>
<td>6.0</td>
<td>45.6</td>
<td>20.1</td>
<td>9.7</td>
</tr>
<tr>
<td><strong>sha256 computation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
h32       | 1.2     | 1.1     | 5.4         | 3.5    | 2.0    |
h128      | 9.6     | 8.1     | 28.0        | 18.0   | 14.6   |
h512      | 50.8    | 49.2    | 113.7       | 90.6   | 65.1   |
h1024     | 179.0   | 165.2   | 280.9       | -      | 174.0  |

### Table 4: Proof size (bytes)

<table>
<thead>
<tr>
<th>Name</th>
<th>bctv14a</th>
<th>groth16</th>
<th>bulletproof</th>
<th>ligero</th>
<th>aurora</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dynamic memory access</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond10</td>
<td>437</td>
<td>209</td>
<td>4,062</td>
<td>2,090,000</td>
<td>235,351</td>
</tr>
<tr>
<td>cond100</td>
<td>437</td>
<td>209</td>
<td>5,146</td>
<td>13,300,000</td>
<td>410,156</td>
</tr>
<tr>
<td>cond1000</td>
<td>437</td>
<td>209</td>
<td>6,464</td>
<td>-</td>
<td>629,000</td>
</tr>
<tr>
<td><strong>Sorting an array</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>med10</td>
<td>437</td>
<td>209</td>
<td>4,306</td>
<td>3,800,000</td>
<td>250,000</td>
</tr>
<tr>
<td>med100</td>
<td>437</td>
<td>209</td>
<td>5,810</td>
<td>27,100,000</td>
<td>404,300</td>
</tr>
<tr>
<td><strong>sha256 computation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
h32       | 437     | 209     | 4,931       | 7,700,000 | 321,289 |
h128      | 437     | 209     | 5,351       | 14,100,000 | 445,000 |
h512      | 437     | 209     | 5,146       | 26,900,000 | 407,200 |
h1024     | 437     | 209     | 6,015       | -      | 517,000 |
<table>
<thead>
<tr>
<th>Name</th>
<th>BCTV14a</th>
<th>Groth16</th>
<th>BCTV14a</th>
<th>Groth16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dynamic memory access</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond10</td>
<td>1.8</td>
<td>1.5</td>
<td>0.4</td>
<td>0.3</td>
</tr>
<tr>
<td>cond100</td>
<td>134.0</td>
<td>124.0</td>
<td>13.7</td>
<td>10.0</td>
</tr>
<tr>
<td>cond1000</td>
<td>12,595.0</td>
<td>11,878.0</td>
<td>1,017.2</td>
<td>900.7</td>
</tr>
<tr>
<td><strong>Sorting an array</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>med10</td>
<td>3.1</td>
<td>2.3</td>
<td>0.7</td>
<td>0.6</td>
</tr>
<tr>
<td>med100</td>
<td>366.0</td>
<td>293.0</td>
<td>76.2</td>
<td>50.6</td>
</tr>
<tr>
<td><strong>sha256 computation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h32</td>
<td>66.3</td>
<td>61.7</td>
<td>6.2</td>
<td>4.6</td>
</tr>
<tr>
<td>h128</td>
<td>491.0</td>
<td>474.0</td>
<td>32.6</td>
<td>27.6</td>
</tr>
<tr>
<td>h512</td>
<td>2,488.0</td>
<td>2,426.0</td>
<td>154.5</td>
<td>131.1</td>
</tr>
<tr>
<td>h1024</td>
<td>6,481.0</td>
<td>6,379.0</td>
<td>357.9</td>
<td>331.4</td>
</tr>
</tbody>
</table>
Appendix 2: Source code

Here is the source code of the programs we used for the benchmarking. It can also be found on github at the following address:

https://github.com/sikoba/isekai/tree/develop/tests/benchmark1

cond100.c – Dynamic Memory Access

```c
struct Input {
    int a[100];
    int b[100];
};

struct Output {
    int x;
};

static inline __attribute__((always_inline)) int compute(int a[], int b[], int n) {
    int result = 0;
    for(int i = 0; i < n-1; i++)
    {
        if (b[i] < n/2)
            result *= a[b[i]];
        else
            result += a[b[i]];
    }
    return result;
}

void outsource(struct Input *input, struct Output *output)
{
    int n = sizeof(input->a) / sizeof(input->a[0]);
    output->x = compute(input->a, input->b, n);
}
```

med100.c – Median

```c
struct Input {
    int a[101];
};

struct Output {
    int x;
};
```
```c
static inline __attribute__((always_inline)) void swap(int *p, int *q) {
    int t;
    t = *p;
    *p = *q;
    *q = t;
}

static inline __attribute__((always_inline)) void sort(int a[], int n) {
    int i, j;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (a[j] > a[j+1])
                swap(&a[j], &a[j+1]);
        }
    }
}

void outsource(struct Input *input, struct Output *output) {
    int n = sizeof(input->a) / sizeof(input->a[0]);
    sort(input->a, n);
    output->x = input->a[n/2];
}
```

---

**hash.cpp – Sha256**

```c
#include <string.h>
#include <cstdio>

#define SHA2_SHFR(x, n) (x >> n)
#define SHA2_ROTR(x, n) ((x >> n) | (x << ((sizeof(x) << 3) - n)))
#define SHA2_ROTL(x, n) ((x << n) | (x >> ((sizeof(x) << 3) - n)))
#define SHA2_CH(x, y, z) ((x & y) ^ (~x & z))
#define SHA2_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
#define SHA256_F1(x) (SHA2_ROTR(x, 2) ^ SHA2_ROTR(x, 13) ^ SHA2_SHFR(x, 22))
#define SHA256_F2(x) (SHA2_ROTR(x, 6) ^ SHA2_ROTR(x, 11) ^ SHA2_ROTR(x, 25))
#define SHA256_F3(x) (SHA2_ROTR(x, 7) ^ SHA2_ROTR(x, 18) ^ SHA2_SHFR(x, 3))
#define SHA256_F4(x) (SHA2_ROTR(x, 17) ^ SHA2_ROTR(x, 19) ^ SHA2_SHFR(x, 10))

#define SHA2_UNPACK32(x, str) { 
    *((str) + 3) = (uint8) ((x) ); 
    *((str) + 2) = (uint8) ((x) >> 8); 
    *((str) + 1) = (uint8) ((x) >> 16); 
    *((str) + 0) = (uint8) ((x) >> 24); 
}

#define SHA2_PACK32(str, x) {
    *(x) = ((uint32) *((str) + 3) ) | ((uint32) *((str) + 2) << 8) 
    \```
```
24    \((\text{uint32} \times ((\text{str}) + 1) \ll 16) \)
25    \((\text{uint32} \times ((\text{str}) + 0) \ll 24)); \}
26
27 #define SHA224_256_BLOCK_SIZE 64
28 #define DIGEST_SIZE 32
29
30 struct Input {
31    char a[32];
32};
33
34 struct Output {
35    unsigned char digest[DIGEST_SIZE];
36};
37
38 class SHA256
39 {
40    protected:
41        typedef unsigned char uint8;
42        typedef unsigned int uint32;
43        typedef unsigned long long uint64;
44
45        uint32 sha256_k[64];
46
47    public:
48        __attribute__((always_inline)) SHA256()
49        {
50            sha256_k[0] = 0x428a2f98; sha256_k[1] = 0x71374491; sha256_k[2] = 0xb5c0fbcf;
51                \(\leftarrow\) sha256_k[3] = 0xe9b5dba5;
52            sha256_k[4] = 0x3956c25b; sha256_k[5] = 0x71374491; sha256_k[6] = 0xb5c0fbcf;
53                \(\leftarrow\) sha256_k[7] = 0xaab10f75;
54            sha256_k[8] = 0x498dbbfa; sha256_k[9] = 0x01a08707; sha256_k[10] = 0xb361c906;
55                \(\leftarrow\) sha256_k[11] = 0xe9b5dba5;
56            sha256_k[12] = 0x01a08707; sha256_k[13] = 0xb361c906; sha256_k[14] = 0xe9b5dba5;
57                \(\leftarrow\) sha256_k[15] = 0x01a08707;
58            sha256_k[16] = 0x01a08707; sha256_k[17] = 0x01a08707; sha256_k[18] = 0x01a08707;
59                \(\leftarrow\) sha256_k[19] = 0x01a08707;
60            sha256_k[20] = 0x01a08707; sha256_k[21] = 0x01a08707; sha256_k[22] = 0x01a08707;
61                \(\leftarrow\) sha256_k[23] = 0x01a08707;
62            sha256_k[24] = 0x01a08707; sha256_k[25] = 0x01a08707; sha256_k[26] = 0x01a08707;
63                \(\leftarrow\) sha256_k[27] = 0x01a08707;
64            sha256_k[28] = 0x01a08707; sha256_k[29] = 0x01a08707; sha256_k[30] = 0x01a08707;
65                \(\leftarrow\) sha256_k[31] = 0x01a08707;
66            sha256_k[32] = 0x01a08707; sha256_k[33] = 0x01a08707; sha256_k[34] = 0x01a08707;
67                \(\leftarrow\) sha256_k[35] = 0x01a08707;
68            sha256_k[36] = 0x01a08707; sha256_k[37] = 0x01a08707; sha256_k[38] = 0x01a08707;
69                \(\leftarrow\) sha256_k[39] = 0x01a08707;
70            sha256_k[40] = 0x01a08707; sha256_k[41] = 0x01a08707; sha256_k[42] = 0x01a08707;
71                \(\leftarrow\) sha256_k[43] = 0x01a08707;
72            sha256_k[44] = 0x01a08707; sha256_k[45] = 0x01a08707; sha256_k[46] = 0x01a08707;
73                \(\leftarrow\) sha256_k[47] = 0x01a08707;
74            sha256_k[48] = 0x01a08707; sha256_k[49] = 0x01a08707; sha256_k[50] = 0x01a08707;
75                \(\leftarrow\) sha256_k[51] = 0x01a08707;
76            sha256_k[52] = 0x01a08707; sha256_k[53] = 0x01a08707; sha256_k[54] = 0x01a08707;
77                \(\leftarrow\) sha256_k[55] = 0x01a08707;
78            sha256_k[56] = 0x01a08707; sha256_k[57] = 0x01a08707; sha256_k[58] = 0x01a08707;
79                \(\leftarrow\) sha256_k[59] = 0x01a08707;
80            sha256_k[60] = 0x01a08707; sha256_k[61] = 0x01a08707; sha256_k[62] = 0x01a08707;
81                \(\leftarrow\) sha256_k[63] = 0x01a08707;
82
83 m_h[0] = 0x6a09e667;
m_h[1] = 0xbb67ae85;
m_h[2] = 0x3c6ef372;
m_h[3] = 0xa54ff53a;
m_h[4] = 0x510e527f;
m_h[5] = 0xb905688c;
m_h[6] = 0xff8399ab;
m_h[7] = 0x5be0cd19;
m_len = 0;
m_tot_len = 0;

void update(const unsigned char *message, unsigned int len);
void final(unsigned char *digest);

protected:
void transform(const unsigned char *message, unsigned int block_nb);
unsigned int m_tot_len;
unsigned int m_len;
unsigned char m_block[2*SHA224_256_BLOCK_SIZE];
uint32 m_h[8];

extern "C" {
    void outsource(Input *, Output *);
    extern void _unroll_hint(unsigned);
}

inline __attribute__((always_inline)) void SHA256::transform(const unsigned char *message, unsigned int block_nb)
{  
    uint32 w[64];
    uint32 wv[8];
    uint32 t1, t2;
    const unsigned char *sub_block;
    int i;
    int j;
    for (i = 0; i < (int) block_nb; i++) {
        sub_block = message + (i << 6);
        for (j = 0; j < 16; j++) {
            SHA2_PACK32(&sub_block[j << 2], &w[j]);
        }
        for (j = 16; j < 64; j++) {
            w[j] = SHA256_F4(w[j - 2]) + w[j - 7] + SHA256_F3(w[j - 15]) + w[j - 16];
        }
        for (j = 0; j < 8; j++) {
            wv[j] = m_h[j];
        }
        for (j = 0; j < 64; j++) {
            t1 = wv[7] + SHA256_F2(wv[4]) + SHA256_CH(wv[4], wv[5], wv[6])
                + sha256_k[j];
            t2 = SHA256_F1(wv[0]) + SHA256_MAJ(wv[0], wv[1], wv[2]);
            wv[7] = wv[6];
            wv[6] = wv[5];
            wv[5] = wv[4];
            wv[3] = wv[2];
            wv[2] = wv[1];
            wv[1] = wv[0];
            wv[0] = t1 + t2;
        }
    }
}
for (j = 0; j < 8; j++) {
    m_h[j] += wv[j];
}

inline __attribute__((always_inline)) void * memcopy( unsigned char * destination, const unsigned char * source, size_t num ) {
    for (int i = 0; i < num; ++i) {
        *(destination+i) = *(source+i);
    }
    return destination;
}

inline __attribute__((always_inline)) void SHA256::update(const unsigned char *message, unsigned int len)
{
    unsigned int block_nb;
    unsigned int new_len, rem_len, tmp_len;
    const unsigned char *shifted_message;
    tmp_len = SHA224_256_BLOCK_SIZE - m_len;
    rem_len = len < tmp_len ? len : tmp_len;
    memcopy(&m_block[m_len], message, rem_len);
    if (m_len + len < SHA224_256_BLOCK_SIZE) {
        m_len += len;
        return;
    }
    new_len = len - rem_len;
    block_nb = new_len / SHA224_256_BLOCK_SIZE;
    shifted_message = message + rem_len;
    transform(m_block, block_nb);
    permanentize(shifted_message, block_nb);
    rem_len = new_len % SHA224_256_BLOCK_SIZE;
    memcopy(m_block, &shifted_message[block_nb << 6], rem_len);
    m_len = rem_len;
    m_tot_len += (block_nb + 1) << 6;
}

inline __attribute__((always_inline)) void SHA256::final(unsigned char *digest)
{
    unsigned int block_nb;
    unsigned int pm_len;
    unsigned int len_b;
    int i;
    block_nb = ((SHA224_256_BLOCK_SIZE - 9) < (m_len % SHA224_256_BLOCK_SIZE));
    len_b = m_tot_len + m_len) << 3;
    pm_len = block_nb << 6;
    memset(m_block + m_len, 0, pm_len - m_len);
    for (int i=0; i < pm_len - m_len; ++i) {
        (m_block + m_len)[i] = 0;
    }
    m_block[m_len] = 0x80;
    SHA2_UNPACK32(len_b, m_block + pm_len - 4);
    transform(m_block, block_nb);
    for (i = 0; i < 8; i++) {
        SHA2_UNPACK32(m_h[i], &digest[i << 2]);
    }
void outsource(struct Input *input, struct Output *output)
{
    unsigned char digest[DIGEST_SIZE];
    //memset(digest, 0, SHA256::DIGEST_SIZE);
    for (int i = 0; i < DIGEST_SIZE; i++)
        digest[i] = 0;

    SHA256 ctx;
    int input_size = sizeof(input->a) / sizeof(input->a[0]);
    ctx.update((unsigned char*)input->a, input_size);
    ctx.final(digest);

    char buf[2*DIGEST_SIZE+1];
    buf[2*DIGEST_SIZE] = 0;
    for (int i = 0; i < DIGEST_SIZE; i++)
    {
        buf[i] = 0;
        buf[i+1] = 0;
        // sprintf(buf+i*2, "%02x", digest[i]);
        output->digest[i] = digest[i];
    }

    //return std::string(buf);