sikoba RESEARCH

Benchmarking Zero-Knowledge Proofs with isekai *

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

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.

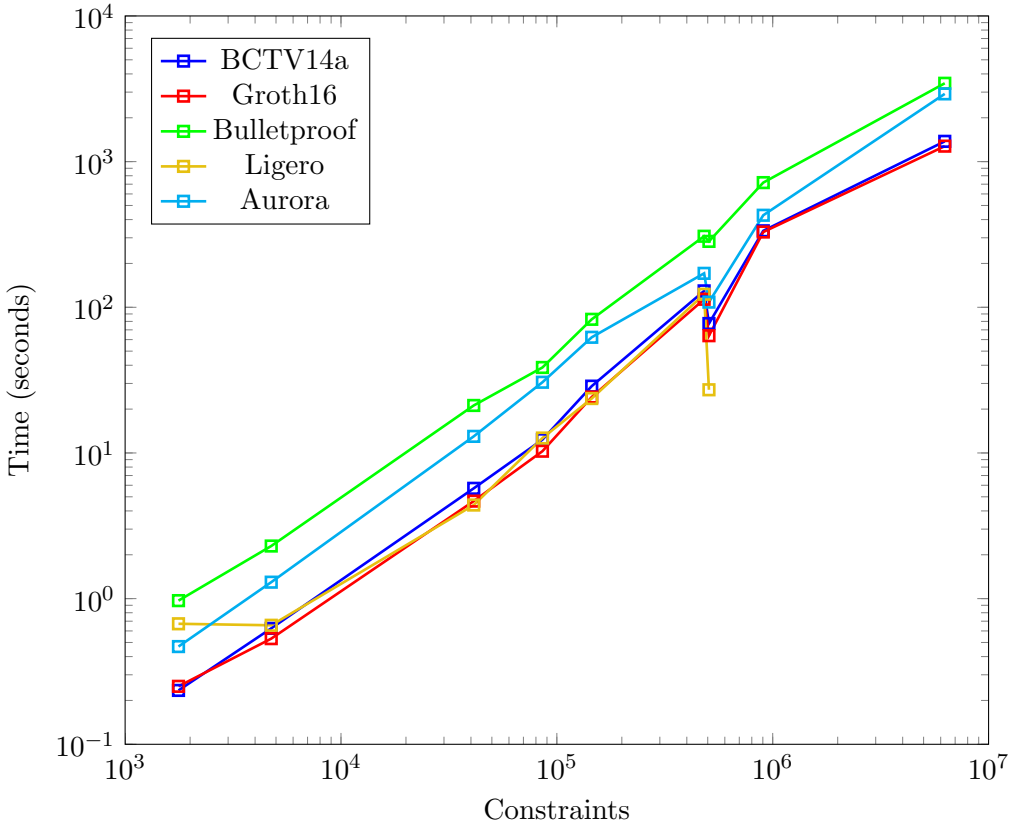
The computer used was a Lenovo T580 laptop with an i7–8550U processor (base frequency @ 1.80 GHz, turbo @ 4.00 GHz), 32 GB RAM and a 1 TB SSD hard drive. Note that for computations involving the most constraints, the entire memory was used and the system had to swap, which obviously affected performance.

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.

Plot 1: Proof Time

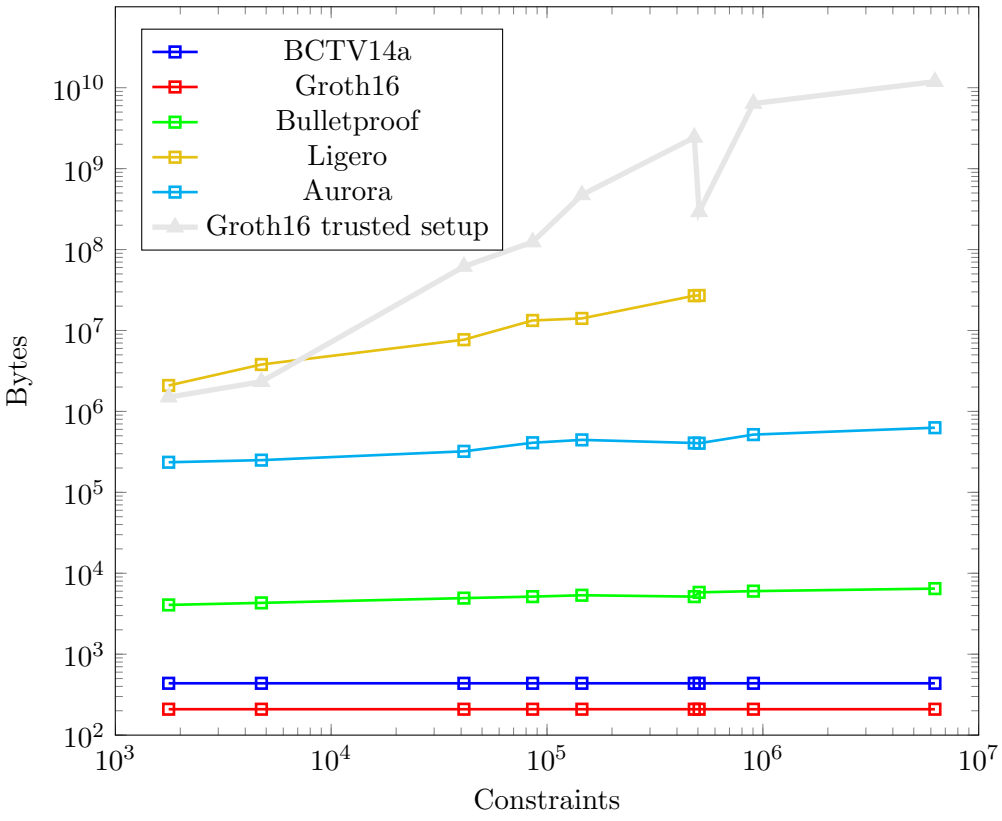


The graph does not always show an increase, meaning that the number of constraints is not the only factor affecting performance. The complexity of the constraints is also important.

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.

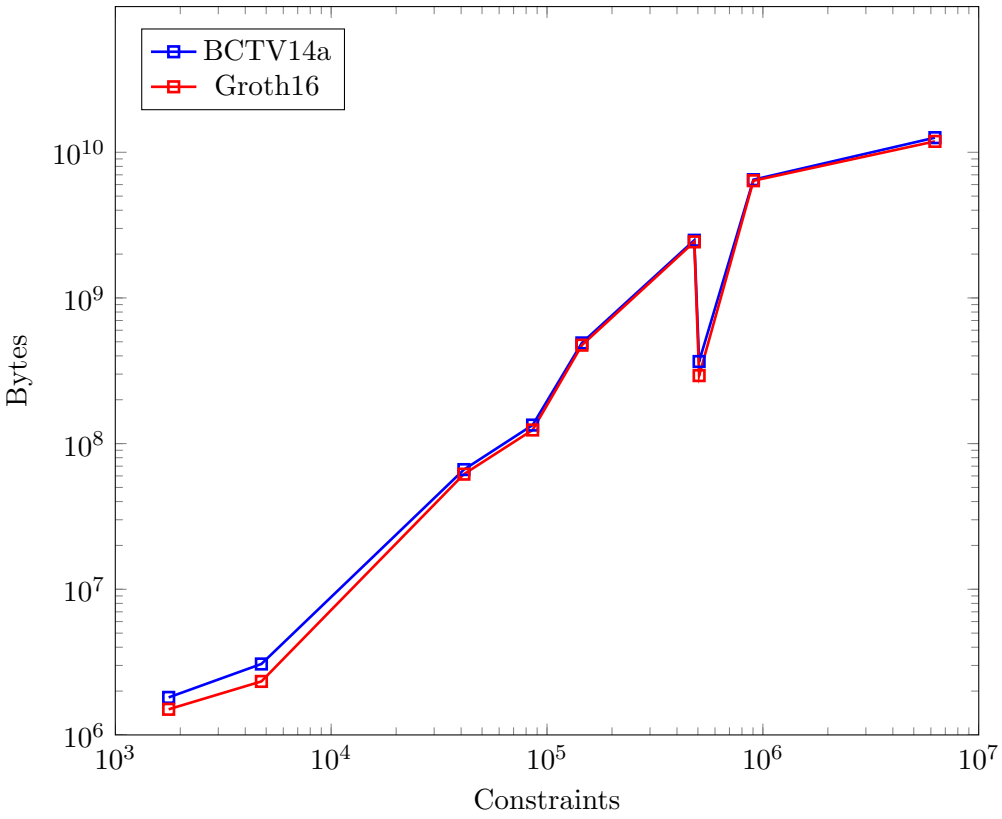
Plot 2: Proof Size vs. Trusted Setup



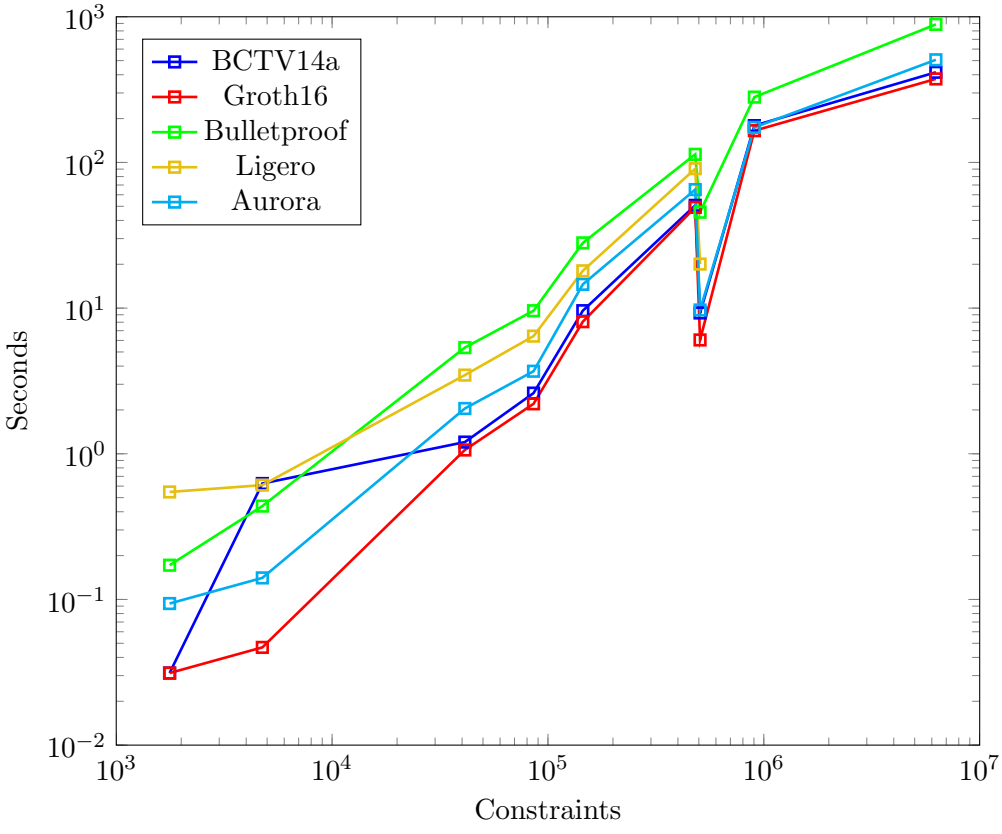
The proof size of zk-SNARKs is extremely small and constant: 209 bytes for the Groth16 scheme and 437 bytes for BCTV14a. However, the size of the trusted setup grows rapidly, up to 12 gb for 6 million constraints. Note that this is the raw data, and it turns out that this data compresses very well. One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



Plot 4: Verification Time



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

[Cry] Crystal language. https://crystal-lang.org/.

[Fan] Fantom foundation. https://fantom.foundation.

[ise] isekai. https://github.com/sikoba/isekai.

[LLV] Llvm. https://llvm.org.

[Sik] Sikoba research. https://research.sikoba.com.

Appendix 1: Benchmarking Data

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

Table 1: Circuit and R1CS

Name	Circuit*	R1CS*	Constraints	Witnesses
Dynamic memory access				
cond10	0.2	0.4	1,767	1,745
cond100	6.7	40.8	85,745	85,461
cond1000	600.0	4,181.0	6,266,234	6,263,354
Sorting an array				
med10	0.2	0.4	4,752	4,646
med100	24.1	77.2	506,502	495,326
sha256 computation				
h32	4.2	55.6	41,210	40,958
h128	13.0	502.0	145,061	144,403
h512	43.3	2,640.0	481,091	478,906
h1024	85.5	7,060.0	904,058	899,843

** size in MB*

Table 2: Proof time (seconds)

Name	bctv14a	groth16	bulletproof	ligero	aurora
Dynamic memory access					
cond10	0.2	0.3	1.0	0.7	0.5
cond100	12.3	10.3	38.6	12.7	30.6
cond1000	1,378.2	1,277.0	3450.0	-	2,918.5
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)

Name	bctv14a	groth16	bulletproof	ligero	aurora
Dynamic memory access					
cond10	0.0	0.0	0.2	0.5	0.1
cond100	2.6	2.2	9.6	6.4	3.7
cond1000	416.0	374.5	887.0	-	506.6
Sorting an array					
med10	0.6	0.0	0.4	0.6	0.1
med100	9.3	6.0	45.6	20.1	9.7
sha256 computation					
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)

Name	bctv14a	groth16	bulletproof	ligero	aurora
Dynamic memory access					
cond10	437	209	4,062	2,090,000	235,351
cond100	437	209	5,146	13,300,000	410,156
cond1000	437	209	6,464	-	629,000
Sorting an array					
med10	437	209	4,306	3,800,000	250,000
med100	437	209	5,810	27,100,000	404,300
sha256 computation					
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 5: Trusted Setup

	Size (MB)		Time (seconds)	
Name	**BCTV14a**	**Groth16**	**BCTV14a**	**Groth16**
Dynamic memory access				
cond10	1.8	1.5	0.4	0.3
cond100	134.0	124.0	13.7	10.0
cond1000	12,595.0	11,878.0	1,017.2	900.7
Sorting an array				
med10	3.1	2.3	0.7	0.6
med100	366.0	293.0	76.2	50.6
sha256 computation				
h32	66.3	61.7	6.2	4.6
h128	491.0	474.0	32.6	27.6
h512	2,488.0	2,426.0	154.5	131.1
h1024	6,481.0	6,379.0	357.9	331.4

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

```
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

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


};

struct Output {
    int x;
};
```

```
10
11 static inline __attribute__((always_inline)) void swap(int *p,int *q) {
12    int t;
13
14    t=*p;
15    *p=*q;
16    *q=t;
17 }
18
19 static inline __attribute__((always_inline)) void sort(int a[],int n) {
20    int i,j;
21
22    for(i = 0;i < n-1;i++) {
23       for(j = 0;j < n-i-1;j++) {
24          if(a[j] > a[j+1])
25                  {
26                          swap(&a[j],&a[j+1]);
27                  }
28       }
29    }
30 }
31
32 void outsource(struct Input *input, struct Output *output)
33 {
34     int n=sizeof(input->a) / sizeof(input->a[0]);
35
36         sort(input->a,n);
37
38     output->x = input->a[n/2];
39 }
```

hash.cpp – Sha256

```
1  #include <string.h>
2  #include <cstdio>
3
4  #define SHA2_SHFR(x, n)    (x >> n)
5  #define SHA2_ROTR(x, n)   ((x >> n) | (x << ((sizeof(x) << 3) - n)))
6  #define SHA2_ROTL(x, n)   ((x << n) | (x >> ((sizeof(x) << 3) - n)))
7  #define SHA2_CH(x, y, z)  ((x & y) ^ (~x & z))
8  #define SHA2_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
9  #define SHA256_F1(x) (SHA2_ROTR(x,  2) ^ SHA2_ROTR(x, 13) ^ SHA2_ROTR(x, 22))
10 #define SHA256_F2(x) (SHA2_ROTR(x,  6) ^ SHA2_ROTR(x, 11) ^ SHA2_ROTR(x, 25))
11 #define SHA256_F3(x) (SHA2_ROTR(x,  7) ^ SHA2_ROTR(x, 18) ^ SHA2_SHFR(x,  3))
12 #define SHA256_F4(x) (SHA2_ROTR(x, 17) ^ SHA2_ROTR(x, 19) ^ SHA2_SHFR(x, 10))
13 #define SHA2_UNPACK32(x, str)                 \
14 {                                             \
15     *((str) + 3) = (uint8) ((x)      );       \
16     *((str) + 2) = (uint8) ((x) >>  8);       \
17     *((str) + 1) = (uint8) ((x) >> 16);       \
18     *((str) + 0) = (uint8) ((x) >> 24);       \
19 }
20 #define SHA2_PACK32(str, x)                   \
21 {                                             \
22     *(x) =   ((uint32) *((str) + 3)      )    \
23            | ((uint32) *((str) + 2) <<  8)    \
```

```
24              | ((uint32) *((str) + 1) << 16)    \
25              | ((uint32) *((str) + 0) << 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
39  class SHA256
40  {
41  protected:
42      typedef unsigned char uint8;
43      typedef unsigned int uint32;
44      typedef unsigned long long uint64;
45
46      uint32 sha256_k[64];
47
48  public:
49      __attribute__((always_inline)) SHA256()
50      {
51          sha256_k[0] = 0x428a2f98; sha256_k[1] = 0x71374491; sha256_k[2] = 0xb5c0fbcf;
             ↪  sha256_k[3] = 0xe9b5dba5;
52          sha256_k[4] = 0x3956c25b; sha256_k[5] = 0x59f111f1; sha256_k[6] = 0x923f82a4;
             ↪  sha256_k[7] = 0xab1c5ed5;
53          sha256_k[8] = 0xd807aa98; sha256_k[9] = 0x12835b01; sha256_k[10] = 0x243185be;
             ↪  sha256_k[11] = 0x550c7dc3;
54          sha256_k[12] = 0x72be5d74; sha256_k[13] = 0x80deb1fe; sha256_k[14] = 0x9bdc06a7;
             ↪  sha256_k[15] = 0xc19bf174;
55          sha256_k[16] = 0xe49b69c1; sha256_k[17] = 0xefbe4786; sha256_k[18] = 0xfc19dc6;
             ↪  sha256_k[19] = 0x240ca1cc;
56          sha256_k[20] = 0x2de92c6f; sha256_k[21] = 0x4a7484aa; sha256_k[22] = 0x5cb0a9dc;
             ↪  sha256_k[23] = 0x76f988da;
57          sha256_k[24] = 0x983e5152; sha256_k[25] = 0xa831c66d; sha256_k[26] = 0xb00327c8;
             ↪  sha256_k[27] = 0xbf597fc7;
58          sha256_k[28] = 0xc6e00bf3; sha256_k[29] = 0xd5a79147; sha256_k[30] = 0x6ca6351;
             ↪  sha256_k[31] = 0x14292967;
59          sha256_k[32] = 0x27b70a85; sha256_k[33] = 0x2e1b2138; sha256_k[34] = 0x4d2c6dfc;
             ↪  sha256_k[35] = 0x53380d13;
60          sha256_k[36] = 0x650a7354; sha256_k[37] = 0x766a0abb; sha256_k[38] = 0x81c2c92e;
             ↪  sha256_k[39] = 0x92722c85;
61          sha256_k[40] = 0xa2bfe8a1; sha256_k[41] = 0xa81a664b; sha256_k[42] = 0xc24b8b70;
             ↪  sha256_k[43] = 0xc76c51a3;
62          sha256_k[44] = 0xd192e819; sha256_k[45] = 0xd6990624; sha256_k[46] = 0xf40e3585;
             ↪  sha256_k[47] = 0x106aa070;
63          sha256_k[48] = 0x19a4c116; sha256_k[49] = 0x1e376c08; sha256_k[50] = 0x2748774c;
             ↪  sha256_k[51] = 0x34b0bcb5;
64          sha256_k[52] = 0x391c0cb3; sha256_k[53] = 0x4ed8aa4a; sha256_k[54] = 0x5b9cca4f;
             ↪  sha256_k[55] = 0x682e6ff3;
65          sha256_k[56] = 0x748f82ee; sha256_k[57] = 0x78a5636f; sha256_k[58] = 0x84c87814;
             ↪  sha256_k[59] = 0x8cc70208;
66          sha256_k[60] = 0x90befffa; sha256_k[61] = 0xa4506ceb; sha256_k[62] = 0xbef9a3f7;
             ↪  sha256_k[63] = 0xc67178f2;
67
68          m_h[0] = 0x6a09e667;
```

```
69          m_h[1] = 0xbb67ae85;
70          m_h[2] = 0x3c6ef372;
71          m_h[3] = 0xa54ff53a;
72          m_h[4] = 0x510e527f;
73          m_h[5] = 0x9b05688c;
74          m_h[6] = 0x1f83d9ab;
75          m_h[7] = 0x5be0cd19;
76
77          m_len = 0;
78          m_tot_len = 0;
79      }
80
81          void update(const unsigned char *message, unsigned int len);
82      void final(unsigned char *digest);
83
84  protected:
85          void transform(const unsigned char *message, unsigned int block_nb);
86      unsigned int m_tot_len;
87      unsigned int m_len;
88      unsigned char m_block[2*SHA224_256_BLOCK_SIZE];
89      uint32 m_h[8];
90  };
91
92   extern "C" {
93      void outsource(Input *, Output *);
94     extern void _unroll_hint(unsigned);
95  };
96
97  inline __attribute__((always_inline)) void SHA256::transform(const unsigned char *message,
    ↪  unsigned int block_nb)
98  {
99      uint32 w[64];
100     uint32 wv[8];
101     uint32 t1, t2;
102     const unsigned char *sub_block;
103     int i;
104     int j;
105     for (i = 0; i < (int) block_nb; i++) {
106         sub_block = message + (i << 6);
107         for (j = 0; j < 16; j++) {
108             SHA2_PACK32(&sub_block[j << 2], &w[j]);
109         }
110         for (j = 16; j < 64; j++) {
111             w[j] =  SHA256_F4(w[j -  2]) + w[j -  7] + SHA256_F3(w[j - 15]) + w[j - 16];
112         }
113         for (j = 0; j < 8; j++) {
114             wv[j] = m_h[j];
115         }
116         for (j = 0; j < 64; j++) {
117             t1 = wv[7] + SHA256_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
118                 + sha256_k[j] + w[j];
119             t2 = SHA256_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
120             wv[7] = wv[6];
121             wv[6] = wv[5];
122             wv[5] = wv[4];
123             wv[4] = wv[3] + t1;
124             wv[3] = wv[2];
125             wv[2] = wv[1];
126             wv[1] = wv[0];
127             wv[0] = t1 + t2;
128         }
```

```
129         for (j = 0; j < 8; j++) {
130             m_h[j] += wv[j];
131         }
132     }
133 }
134 
135 inline __attribute__((always_inline)) void * memcopy( unsigned char * destination, const
    ↪  unsigned char * source, size_t num )
136 {
137         for (int i = 0 ; i < num ; ++i)
138         {
139                 *(destination+i) = *(source+i);
140         }
141         return destination;
142 }
143 
144 
145 inline __attribute__((always_inline)) void SHA256::update(const unsigned char *message,
    ↪  unsigned int len)
146 {
147     unsigned int block_nb;
148     unsigned int new_len, rem_len, tmp_len;
149     const unsigned char *shifted_message;
150     tmp_len = SHA224_256_BLOCK_SIZE - m_len;
151     rem_len = len < tmp_len ? len : tmp_len;
152     memcopy(&m_block[m_len], message, rem_len);
153     if (m_len + len < SHA224_256_BLOCK_SIZE) {
154         m_len += len;
155         return;
156     }
157     new_len = len - rem_len;
158     block_nb = new_len / SHA224_256_BLOCK_SIZE;
159     shifted_message = message + rem_len;
160     transform(m_block, 1);
161     transform(shifted_message, block_nb);
162     rem_len = new_len % SHA224_256_BLOCK_SIZE;
163     memcopy(m_block, &shifted_message[block_nb << 6], rem_len);
164     m_len = rem_len;
165     m_tot_len += (block_nb + 1) << 6;
166 }
167 
168  inline __attribute__((always_inline)) void SHA256::final(unsigned char *digest)
169 {
170     unsigned int block_nb;
171     unsigned int pm_len;
172     unsigned int len_b;
173     int i;
174     block_nb = (1 + ((SHA224_256_BLOCK_SIZE - 9)
175                      < (m_len % SHA224_256_BLOCK_SIZE)));
176     len_b = (m_tot_len + m_len) << 3;
177     pm_len = block_nb << 6;
178     //memset(m_block + m_len, 0, pm_len - m_len);
179         for (int i=0; i< pm_len - m_len; ++i)
180                 (m_block + m_len)[i] = 0;
181 
182     m_block[m_len] = 0x80;
183     SHA2_UNPACK32(len_b, m_block + pm_len - 4);
184     transform(m_block, block_nb);
185     for (i = 0 ; i < 8; i++) {
186         SHA2_UNPACK32(m_h[i], &digest[i << 2]);
187     }
```

```
188 }
189
190
191
192 void outsource(struct Input *input, struct Output *output)
193 {
194
195         unsigned char digest[DIGEST_SIZE];
196     //memset(digest,0,SHA256::DIGEST_SIZE);
197         for (int i = 0; i < DIGEST_SIZE; i++)
198                 digest[i] = 0;
199
200     SHA256 ctx;
201         int input_size=sizeof(input->a) / sizeof(input->a[0]);
202     ctx.update( (unsigned char*)input->a, input_size);
203     ctx.final(digest);
204
205     char buf[2*DIGEST_SIZE+1];
206     buf[2*DIGEST_SIZE] = 0;
207     for (int i = 0; i < DIGEST_SIZE; i++)
208         {
209                 buf[i] = 0;
210                 buf[i+1] =0;
211   //      sprintf(buf+i*2, "%02x", digest[i]);
212                 output->digest[i] = digest[i];
213         }
214
215     //return std::string(buf);
216 }
```

■