ChaCha related 64 bit oriented ARX cipher

Daniel Nager
daniel.nager@gmail.com

January 2024

Abstract

A cipher scheme related to ChaCha [Ber] with the variation of using 64 bit operations instead of 32 bits, and the same 512 bit state size, is presented. We will provide strong argumentation to assert that the same security of ChaCha can be obtained with half number of instructions for the same number of 20 rounds. Also, an strategy to implement this cipher on SIMD extensions is presented, with a maximal throughput of about 3.37 bytes per cycle on a 256 bit SIMD extension with at least 11 vector registers.

1 Introduction

The point of this document is to present an ARX algorithm that satisfies the intuitive notion that using 64 bit instructions gives the same result with half instructions than using 32 bit instructions. Although, this is not obvious per se, and some changes in the structure of the cipher must be done to achieve this results. The simple idea is that a single 64 bit instruction does the work of two 32 bit instructions so we need half of them. This can be profited in several ways in different scenarios to achieve greater throughput of bytes, or in the worst case to achieve the same throughput, but never more.

Confusion and diffusion are the same in ChaCha, as the key and what we can call plaintext are part of the state. We will use TestU01's Crush [LS] battery of test to check that confusion and diffusion can be considerated the same in ChaCha and in the ARX cipher presented here. This is just a check since we will rationally argue that this is the case further on this document.

To finish this introduction let's state that the confusion and diffusion in ARX ciphers is achieved mainly by the rotation instructions, with no diffusion or confusion from the xor and some residual diffusion and confusion in addition. We assume as clear that each rotation roughly doubles the number of bits diffused and confused, which is coherent with actual number of bits changed in a round of ChaCha changing with respect to a changed bit in the previous state. As a result we conclude that rotating once a 64 bit word that already has 32 bits affected results, with a single rotation, in 64 bits affected, simplifying.

2 The algorithm

Here's a C programming language implementation of the algorithm, necessary to understand next sections of this document:

```
#include <stdint.h>

#define rot(d,v) {v=v<<d|v>>(sizeof(uint64_t)*8-d);}

// this is the 8 64-bit words state
uint64_t a,b,c,d,e,f,g,h;
uint64_t a0,b0,c0,d0,e0,f0,g0,h0=0;

#define QR(a,b,c,d){\
        a+=b;d^=c;rot(43,a);\
        c+=a;b^=d;rot(17,c);}

#define ROUNDS 20

void do_ARX()
{
        int i;

        a=a0;b=b0;c=c0;d=d0;
        e=e0;f=f0;g=g0;h=++h0;

        for(i=0;i<ROUNDS;i++){
                QR(a,b,c,d);
                QR(e,f,g,h);
                QR(a,b,e,f);
                QR(c,d,g,h);
        }

        a^=a0;b^=b0;c^=c0;d^=d0;
        e^=e0;f^=f0;h^=h0;
}
```

3 Number of rotations

As we've stated above half number of instructions are executed in this code QR quarter round than in the QR of ChaCha. We need to examine the four quarter rounds in the main loop to see, this is clear from the code, that as we're using just 8 state words of 64 bits instead of 16 state words of 32 bits, we can pass to the quarter round each state word twice, resulting in the same number of rotations per round. This is a strong argumentation, joint with the fact that 32 bit diffusion is just a rotation away from 64 bit diffusion, to be able to assert that expected diffusion and confusion, and security in general in 20 rounds is the same in this cipher than in ChaCha.

Actually we're considering the state as a rectangular 4×2 matrix:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \end{pmatrix}$$

First two lines of a round mixes state in two rows:

$$\begin{matrix} (a & b & c & d) \\ (e & f & g & h) \end{matrix}$$

Last two lines of a round mixes state, not in columns but in two submatrices:

$$\begin{pmatrix} a & b \\ e & f \end{pmatrix} \begin{pmatrix} c & d \\ g & h \end{pmatrix}$$

So actually we're doing the same number of rotations in this ARX 64-bit oriented cipher than in ChaCha, with half number of instructions, and providing presumably equivalent security.

Let's comment that TestU01's Crush is passed both for this cipher in 4 rounds and in ChaCha for 4 rounds, as expected.

4 Superscalarity and Single Instruction Multiple Data

The quarter round is arranged in a way that every adjacent instruction can be executed by a 2-superscalar processor at once. So the quarter round uses 3 processor cycles. We execute a total of 4 quarter rounds in 20 rounds, so the amount of cycles is $3 \cdot 4 \cdot 20 = 240$ cycles to generate 64 bytes. This is at most 3.75 cycles/byte in a modern computer without using SIMD, amounting to a total of about 4 cycles/byte with state initialization and final xor.

In SIMD, as the state is just 8 words, there's no difficulty, if we consider the register set of a SIMD extension as a matrix, where registers are rows, to do parallel block execution of the cipher putting parallel states straight on columns. So if we target a 256 bits SIMD we only need to use 10 registers if rotation must be done with two shifts and a xor to process a total of 4 blocks, resulting in a resulting amount of $4 * 64 = 256$ bytes of cipher stream data. We can count the number of instructions executed that are 10 for the quarter round, 4 times per round and for 20 rounds results into $10 \cdot 4 \cdot 20 = 800$ SIMD instructions, with the outcome of 3.125 cycles per byte excluding any auxiliary initialization and post processing that will be treated in next section.

In other SIMD scenarios the throughput is no so high but let's note first that this approach can be applied to almost every SIMD extension and second that no permuting of elements in vector registers are needed, just applying sequentially additions, xors and rotations as defined by the algorithm.

5 Pre and post processing in Single Instruction Multiple Data

Let's take as an example a SIMD extension with at least 11 of 256 bits, which corresponds to 4 64-bits registers.

First we need an initial setup of the whole stream generation. In memory we set up a matrix where rows are SIMD registers and columns are initial states. All columns are identical except the entry corresponding to the block counter.

Initially we load to the register 11th, as we need 10 registers to generate the cipher stream, an incremental counter in the range $[-4, \ldots, -1]$. This is the setup. Next we apply the following procedure:

1. We add 4 to every word in the 11th register which holds the counter and we copy it to the corresponding register of the first 8, that's where we will do the mixing.
2. We copy from memory or from spare registers the initial state to the rest of the first 8 registers.
3. We do the mixing as explained in previous section.
4. We load from memory and xor the initial states and counters to the first 8 registers.
5. We save to a memory location the 8 first registers that hold 4 blocks of stream data.
6. We do the transpose of the matrix in memory, whose elements are 64-bit words, in order to get 4 consecutive blocks with incremental counter. This data is ready to be used as stream data with no further changes.

So in the long run we do 7 moves from memory or registers to registers, an update of the counters, a move of the counters, load 7 registers from memory and xor it, along with the counter, to 8 registers, a write back of 8 registers to memory and a 4×8 memory matrix transpose, amounting to 32 SIMD instructions and 30 memory move instructions to get 4 stream blocks. This procedure adds 0.2421 cycles per byte approximately in this scenario, while mixing takes 3.125 cycles per byte as explained previously and resulting in a total of 3.3671 cycles per byte.

References

[Ber] Daniel J. Berstein. *ChaCha, a variant of Salsa20.* URL: `https://cr.yp.to/chacha/chacha-20080128.pdf`.

[LS] Pierre L'Ecuyer and Richard Simard. *TestU01: A C Library for Empirical Testing of Random Number Generators.* URL: `https://www.iro.umontreal.ca/~lecuyer/myftp/papers/testu01.pdf`.