Assignment Report

Felix Mölder s1022118
Cristian Bassotto s1148245

January 17, 2025

1 ChaCha20 Optimization

For this implementation, we looked at the general structure of ChaCha20. It consists of 20 rounds. Each round consists of 4 quarterround functions. In turn, each of this function consists of 4 rotations, 4 additions and 4 XORs. To optimize the implementation we started by investigate the rotation function.

1.1 The rotate function:

This function rotates a value a of size n by a certain amount d to the left. In C, this is achieved by saving the first $n - d$ bits of a, say t, Shift a by d bits to the right and concatenate both together. Doing this instructions in Assembly would several instructions. However, if we consider rotate left by d, this is the same as doing rotate right by $n - d$ and this can be given by only one instruction:

```
ror     r0, r0, #n-d
```

Thus, we can reduce all instructions necessary for the C-implementation, to this one instruction.

1.2 The function calls:

If we want to optimize the code in Assembly, then function calls are wasting a lot of cycles. We need to push and pull all registers whenever a function is called/returning. For example push and pop 4 registers to the stack will cost $1 + 4 + 1 + 4 = 10$ cycles. We therefore remove all function calls and basically write every assembly line one by one in a sequential order.

1.3 The for loop:

In ChaCha20, 8 quarterrounds (2 full rounds) are done in a for-loop. Since ChaCha20 does 20 rounds, this for-loop will iterate 10 times. As shown in the lecture we will enroll the for loop to minimize the cycles wasted by saving loop iterator variables and others.

1.4 Result:

We finally obtain the Assembly-code as attached in the file and the test gives **27.000 cycles**. We could have used more of the techniques from the slides, but we thought it would be better to continue with the others first.

2 Poly1305 Optimization

For this implementation, we looked into the design of poly1305. We saw that the standard uses radix 8. This means that the internal state of numbers is based on a 8-bit representation, where each chunk of the message is processed in 8-bit segments. In our initial implementation using radix 8, it took 87,482 cycles to process 512 bytes of input.

To optimize performance, we decided to change it to radix-26, which means that we used a 26-bit representation instead of 8-bit. This change allowed us to handle larger chunks of data per operation, reducing the number of cycles required to process the same amount of data to 27091 cycles.

2.1 Translation

However, shifting to a radix 26 representation introduced some challenges, particularly with the translation between the given key k and the input data and the internal representation used during computation. The key k, input *in* and output *out* are passed as pointers to unsigned char. This means that can be directly converted into a 8-bit representation, but they need to be adapted for a 26-bit representation splitting some of the bytes into different chunks.

To adapt them, we have created a shifter array with the starting point of each of the 17 bytes of the 8-bit array in the 26-bit array, which means, for example, that the first item starts from 0, while the second should start from 8 (to leave space for the first one) and so on. With this shifter we could manage to retrieve the exact position where to copy the 8-bit item in the 26-bit item without having to perform a modular operation that might involve more than 2 cycles to complete. To convert the indexes from the 8-bit to the 26-bit representation in order to address the right items in the two arrays, we use a simple calculation: $i = (j * 8)/26$ where j is the array index in the 8-bit while i is the correspondent item in the 26-bit representation.

Moreover, to make the translation constant time, we define a variable that stores the remaining bits to copy in the next 26-bit item. That variable is shifted by $26 - shifter$ in order to be 0 in the case that all bits have already been copied to the 26-bit item, or store only the relevant bits otherwise. This idea basically follows the implementation from the reduction of add.

To translate back to a 8-bit array, we concatenate the 26-bit item, where the byte starts, with the 26-bit item, where next byte should start. Doing so, we manage to take into account if a byte is split into two different items, without compromising the regular case where an item is stored in a single item, while maintaining the computation in constant time.

2.2 Modular Multiplication

A second challenge derive from the reduction in the modular multiplication. In particular, the main point focuses on the scaling factor to ensure the ring property of $2^{130}-5$. In the radix-8 this is computed by $2^{136} \mod (2^{130} - 5) = 320$. However in our case with a total representation of $26 * 5 = 130$ bits, the scaling factor became $2^{130} \mod (2^{130} - 5) = 5$.

3 ECDH Optimization

For ECDH we investigated the source code for any timing leaks. We found out that in smult.c the if statement decides on the secret key. Thus we have a secret-dependent branch which can be used in Side-Channel-Attacks to reconstruct the key. As show in the lectures we first tried to always double and add. In the case of a 1, we add P, otherwise we add $\mathcal{O}$ (the neutral element). Since the performance is worse and the implementation is still not completely constant time, we chose 2 optimization techniques to reduce the number of cycles:

3.1 Fixed-window scalar multiplication

We decided to implement the fixed-window scalar multiplication by setting $\omega = 4$ and therefore we created a precomputation table T with $2^\omega = 16$ elements ($\{\mathcal{O}, P, 2P, \ldots, 15P\}$) by adding P 15 times. This takes 15 additions. However, we reduce the number of additions in the for-loop from 128 times (on average) to 2 times per byte, which is 64 times. Thus we reduced the number of additions from 128 to 79 times (reduction by nearly 40 %) even with the precomputations. Because the lookup table is secret dependent we still do not have a constant time implementation here. We used the definitions from the slides for constant time lookup and constant time cmov, where in cmov we changed the size of for-loop to 16. This implementation gives the correct results and increased the amount of cycles by only around 2.000.000 which is just $\approx 4.4\%$ of the total number of cycles before any optimization technique was used. This makes the fixed-window implementation constant time and efficient. Another windows size can change the efficiency of the implementation (see Further optimization options).s

3.2 Radix optimization

As per the poly1305 optimization, we decide to change the radix representation of the field arithmetic from radix-8 to radix-26. As before we created a shifter array and proceed as described before for translation between 8-bit and 26-bit arrays of dimension 32 and 10. This time we also had to translate the constant values from the byte configuration to the 26-bit configuration.

Apart from the challenges faces also during the implementation of poly1305, this time, we also had to modify the implementation of the modular multiplication. In particular, due to the large radix, we had to reduce also the intermediate results in the multiplication. This is cased by the limited amount of bits that can be stored in a C variable. A variable can have a maximum size of 64-bit in C. However, during the multiplication the values can take up to 66-bit length, that, then, can overflow the maximum capacity of the unsigned long long. To avoid this, we reduce the right part of the intermediate results before multiplying by $2^{260} \mod (2^{255} - 19) = 608$ and adding them to the left intermediate results. An optimized version could include a radix-25.5 where this issue is avoid.

4 Summary and Conclusion

Algorithm	initial impl. in cycles	border in cycles	our impl. in cycles
ChaCha20	32.192	30.000	27.491
Poly1305	87.482	80.000	27.091
ECDH	(45.611.933, 42.491.583)	(35.000.000, 32.000.000)	(15.595.424, 14.634.796)

Algorithm	% comp. to border	% comp. to init.	red. by
ChaCha20	$91,6\%$	$85,4\%$	$14,6\%$
Poly1305	$33,9\%$	$31,0\%$	$69,0\%$
ECDH	$44,6\%, 45,7\%$	$34,2\%, 34,4\%$	$65,8\%, 65,6\%$

As we can see from the table, the radix changes have a big impact on the cycles. Transforming ChaCha20 to Assembly and enrolling loops, remove function calls and optimize instructions reduced the number of cycles by around 15 %. However changing from radix-8 to radix-26 only, reduced the number of cycles by nearly 70 %. Similar we can see with ECDH, where moving from radix-8 to radix-26 in the field arithmetic of ed25519-curves reduced the amount of cycles by around 73,7 %. However, we lost a bit of performance with achieving constant time scalar multiplication such that the final reduction is around 66 %. We conclude that radix change was the most performance improving technique while other methods such as Assembly optimizations are more helpful to get even the last unnecessary cycle removed.

4.1 Further optimization options

Due to time constrains we were not able to apply all optimization techniques. But it is worth to mention two optimizations that would reduce the number of cycles in these implementations and therefore improve the efficiency.

4.1.1 ECDH - fixed-window with $\omega = 5$

We did some calculations to find the ω that reduces the number of additions in the double-and-add approach the most. As the slides suggest a value of 4 or 5 for 256 bit, we have seen that $\omega = 5$ would reduce the number of additions by 5 compared to $\omega = 4$. However, the implementation with $\omega = 4$ was way easier to implement due to the fact that it divides one byte perfectly into 2 4-bit chunks. Therefore, a smart implementation of $\omega = 5$ could reduce the number of cycles even more.

4.1.2 ECDH - The montgomery ladder

The montgomery ladder is another technique to multiply a scalar with a point on a montgomary curve. the performance has been shown to be better than the fixed window approach we used. There is a known timing leak of a similar type as we have seen in this exercise in the montgomary ladder, but approaches have been made to remove that timing leak and the efficiency could have been reduced

(see [Ber12]) by implementing the constant time version of the montgomary ladder instead of the fixed window.

References

[Ber12] Peter Bernstein, Daniel J.; Schwabe. Neon crypto. *Cryptographic Hardware and Embedded Systems – CHES 2012*, 7428(19):320–339, 2012.