ELLIPTIC CURVE PUBLIC KEY GENERATION: OPTIMIZING BIG NUMBER ARITHMETIC IN RADIX REPRESENTATIONS FOR USE IN CURVE25519

Robin Burkhard, Manuel Meinen, Sabina Fischlin, Supraja Sridhara

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Elliptic Curve Diffie-Hellman Key Exchange (ECDHE[1]) is a popular method for establishing shared secrets using public key cryptography. *Curve25519* is widely adopted for ECDHE because of its various advantages like fast modular arithmetic and short but secure public keys. Generally, the public keys generated are ephemeral in nature, and so speed of generation is critical and fast computations are vital.

Public key generation using *Curve25519* requires arithmetic in the field $\mathrm{F}_{2^{255}-19}$. Fast arithmetic in this field is commonly done using so-called radix representations, in particular radix-2^{51} and radix-$2^{25.5}$.

We implemented a public key generation algorithm using *Curve25519* and introduced a new radix representation, radix-2^{17}. We hypothesized that a higher speedup is achievable using the radix-2^{17} representation because of its higher operational intensity as compared to the previously introduced radix representations.

In the process of optimizing the radix-2^{17} implementation we discussed the various grounds for optimizations and the main challenges we faced. By comparing the speedup of the optimized implementations of the various radix representations, we concluded that our hypothesis holds.

1. INTRODUCTION

Motivation. Elliptic Curve Cryptography (ECC) has various applications in security protocols. Amongst others, it may be used for key generation in a Diffie-Hellman key exchange. The Diffie-Hellman protocol is typically used to establish sessions by exchanging ephemeral keys. Generation of such keys may therefore happen often and is required to be fast.

There are several curves one may choose from for ECC[2] [1]. *Curve25519* was first introduced by Daniel J. Bernstein in [2]. Since then it has widely gained in popularity for use in ECDHE. *Curve25519* is standardized for use in TLS[3] v1.3 [3] and is considered to have various advantages which allows it to be faster than other curves. E.g. it allows for fast modular arithmetic due to its small offset from a power of 2[4]. Additionally, the generation of the public key only requires the output of the x-coordinate [2], making it short but still secure.

Curve25519 operates on 255-bit numbers. Such big number operations are not trivial on 64-bit machines as they cannot be directly represented. Radix representations discussed in [4] break the number into smaller limbs, which can then be represented in 64-bit registers. As we will show in section 2, arithmetic using limbs requires many low-level operations, which leaves a lot of room for optimization.

Contribution. We implemented a public key generation algorithm for ECDHE using *Curve25519*. This work introduces the radix-2^{17} representation to perform the arithmetic in the field $\mathrm{F}_{2^{255}-19}$ and compares the performance of the new representation to the radix representations discussed in [4] and other existing literature, namely radix-2^{51} and radix-$2^{25.5}$.

Related work. "Curve25519: new Diffie-Hellman speed records" [2] introduced the *Curve25519* specification and the intuition behind fast field arithmetic. This was then extended in "Sandy2x: New Curve25519 Speed Records" [4] which elaborated on fast field arithmetic in radix-2^{51} and radix-$2^{25.5}$ (see section 2) representations and set speed records for *Curve25519* ECDHE performed on Sandy Bridge and Ivy Bridge processors. The work illustrated the advantages of radix representations and how they could be used efficiently in Assembly code. We extend [4] to formulate the radix-2^{17} representation and contrast the performance of the different radix representations in this work.

Another way to perform big number arithmetic is to use a library such as GMP[5]. GMP exposes a clean interface canonical to the one that is constructed using the radix representations in this work. We compare the runtime of an implementation using GMP with those based on radix representations.

[1] Elliptic Curve Diffie-Hellman Key Exchange
[2] Elliptic Curve Cryptography
[3] Transport Layer Security
[4] The arithmetic is performed in the F_p where $p = 2^{255} - 19$. The small offset is 19.
[5] GNU Multiple Precision Arithmetic Library

2. BACKGROUND

Elliptic Curve Cryptography. In ECC, we perform operations on points, which are located on an elliptic curve. The elliptic curve is defined by a curve equation over a Galois-Field (GF). We use a Montgomery curve defined by the following equation for some prime number p and curve parameters A and B [5]:

$$By^2 = x^3 + Ax^2 + x \mid A, B, x, y \in GF(p)$$

In ECC multiplying a scalar (say a secret key) with a point on the curve is a commonly used computation. This can be done using a series of point doublings and additions similar to the *square-and-multiply* technique for modular exponentiation. A point addition defined for the elliptic curve is closed, turning the curve over the field into a group. This implies that scalar multiplication on a point always results in a point on the curve itself.

Drawing a line through two points on the curve always intercepts the curve on a third point, which reflected on the x-axis produces the result of point addition. Point doubling, is equivalent to adding a point to itself, and the line is then a tangent to the point and its interception with the curve when reflected is the result of point doubling. We illustrate point addition and doubling in Fig. 1. The formulae for point addition are then:

$$x_3 = B\left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - A - x_1 - x_2$$

$$y_3 = \frac{(2x_1 + x_2 + A)(y_2 - y_1)}{x_2 - x_1} - \frac{B(y_2 - y_1)^3}{(x_2 - x_1)^3} - y_1$$

where (x_1, y_1) and (x_2, y_2) are the points to be added and (x_3, y_3) the resulting point. Similarly point doubling can be computed as:

$$x_2' = Bl^2 - A - 2x_1'$$
$$y_2' = (3x_1' + A)l - Bl^3 - y_1',$$
$$\text{with } l = \frac{3{x_1'}^2 + 2Ax_1' + 1}{2By_1'}$$

where (x_1', y_1') is the initial point and (x_2', y_2') the resulting point.

Public key generation algorithm. Since inversion of the scalar multiplication is hard, a public key (*Pk*) can be generated using ECC by multiplying the secret key (*Sk*, interpreted as a scalar) with the base point (G) .

$$Pk = Sk \times G$$

A side channel resistant manner of implementing scalar multiplication is the Montgomery ladder [5], which iterates over the bits of the scalar and performs a point addition and a point doubling. Thus the pseudo-code for the algorithm we implement is:

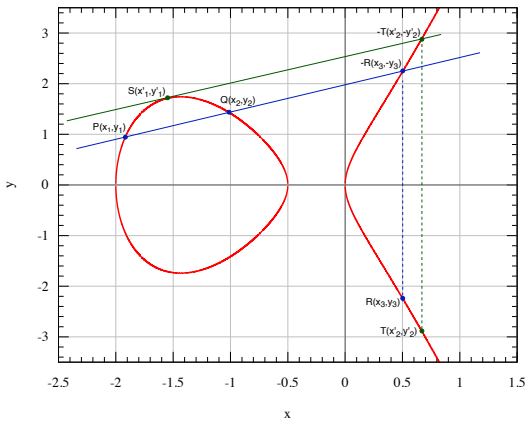

Fig. 1. Illustration of point addition, P+Q = R (blue) and point doubling, 2S=T (green) on a Montgomery curve with A=2.5 and B=0.25. *Extended from figure by Krishnavedala, modified by Florian Weber - Modification of Montgomery_curve1.svg, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=86379477.*

```
R0 <- 0
R1 <- G
for secret_key_bit in Sk:
    if secret_key_bit = 0:
        R1 <- point_add(R0,R1)
        R0 <- point_double(R0)
    else:
        R0 <- point_add(R0, R1)
        R1 <- point_double(R1)
return Pk <- R0
```

***Curve25519* parameters.** We focus our implementation on the elliptic curve defined over GF($2^{255} - 19$) also known as *Curve25519*. The curve was standardized in RFC[6] 7748 [6]. The curve equation is:

$$y^2 = x^3 + 486662x^2 + x$$

We use the standard base point from [2].

Arithmetic in $\mathbf{F_{2^{255}-19}}$ for scalar multiplication on *Curve25519*. To perform scalar multiplication we need the operations addition (*add*), subtraction (*sub*), multiplication (*mult*), square (*sqr*), inverse (*inv*) and division (*div*) in the field $\mathrm{F}_{2^{255}-19}$. This can be done by representing the numbers in a so-called radix-2^r notation, which breaks down a b-bit number into n limbs of at most r bits. The number of limbs n is equal to $\lceil b/r \rceil$. A number f is then represented as:

$$f = \sum_{i=0}^{n-1} f_i 2^{\lceil ir \rceil}.$$

The radix representations can be viewed as polynomials, where the limbs (f_i) are the indeterminates. All operations on the radix-2^r representation are then operations

[6]Request For Comments

on polynomials. This means we can perform the necessary operations as follows:

Add and *sub* are trivial and can be computed limb-wise. The other operations are not straight-forward and require a larger number of low-level (i.e. integer) operations.

The polynomial product can be modified to also directly perform modular reduction in the field $\mathrm{F}_{2^{255}-19}$. An example of such formulae in a radix-2^r representation can be found in [4]. Computing a *sqr* is naturally similar to *mult*, but can be optimized as defined in [7]. *Inv* is computed using Fermat's Little Theorem and the *square-and-multiply* technique for fast modular exponentiation [7]. *Div* operation in the field involves performing an *inv* and a *mult*.

Even though these operations are simple polynomial operations, it is easy to see that the results of each individual limb may then be a number larger than r bits. This means that after each computation we need to perform an additional step, which ensures the number is still a valid radix-2^r representation in field $\mathrm{F}_{2^{255}-19}$. We call this the *reduce* step.

The number of low-level arithmetic operations (and the operational intensity) are directly proportional to the number of limbs in the radix representation. For example, the product of two polynomials requires n^2 multiplications and $n(n^2-1)$ additions. This implies that the choice of r has a direct impact on the operational intensity.

Introducing radix-2^{17} representation. In addition to the radix representations found in literature [4] we introduce radix-2^{17} notation. An integer f modulo $2^{255}-19$ in radix-2^{17} is represented as

$$f = \sum_{i=0}^{14} f_i 2^{\lceil i.17 \rceil}.$$

This representation has 15 limbs and 17 bits/limb. The operations are performed in a manner equivalent to the one described in the previous paragraph. The formula for *mult* is shown in Fig. 2. The formula for *sqr* can be optimized as shown in [7].

Cost Analysis. There are two perspectives we can take with regards to the cost analysis: one is the classical number of low-level integer operations (*iops*). Another is the number of big number operations (*big_num_ops*). Since the number of iops, as seen in the previous paragraphs, differs greatly between the various radix representations, we use the latter to contrast the performance between different radix notations. So we define two cost measures as iops/cycle and big_num_ops/cycle.

To count the *iops* we instrumented our code to maintain a counter for every operation performed and took an average over different secret keys. This was done, as opposed to defining a theoretical measure, since the number of low-level operations depends on the values of the numbers themselves and the number of limbs in the radix representation. Table 1 enumerates the number of 255-bit operations required per computation and is used for the second cost measure. The number of curve computations is equal to the number of bits in the secret key.

	add	sub	mult	sqr	div
double	9	5	11	4	2
add	3	9	5	4	3

Table 1. Number of 255-bit operations per curve computation

3. METHOD

As discussed in the introduction (see section 1), we focused on optimizing the public key generation using the radix-2^{17} representation.

In the following paragraphs we outline each optimization method we applied and provide insight into the techniques and auxiliary means (e.g. the microbenchmarking) we devised during our optimization process.

Hypothesis. As discussed in 2, the operational intensity is directly proportional to the number of limbs. We therefore hypothesized that a significant performance gain – as compared to the other representations – should be achievable by optimizing the radix-2^{17} representation .

Approach. We identified two main areas to optimize:

1. the curve computations (specifically `point_add` and `point_double`)
2. the big number operations (i.e. *add*, *sub*, etc.).

Base implementations. We built a base implementation for each of the radix representations we chose to compare (our own radix-2^{17} and the ones most commonly found in literature, radix-$2^{25.5}$ and radix-2^{51}). Additionally, we wrote an implementation using the GMP library, which we primarily used for validation purposes (see section 4).

All our base implementations were built using the algorithm and the formulae for point addition and doubling on a Montgomery curve as presented in section 2.

Scalar replacement. We performed scalar replacement in the functions for all big number operations (*add*, *sub*, *mult* etc.) where the same variable was accessed more than once. Scalar replacement was most beneficial in the computations in *mult* and *sqr* because each limb is used more than once in the polynomial multiplication formulae.

Function inlining. To inline functions we used the `always_inline` attribute. When compiled with no optimization flags, only these functions were inlined by the compiler. We tried various function inlining combinations and compared the Assembly code generated with no optimizations with the code generated when compiled using the

$$
\begin{aligned}
h_0 &= f_0g_0 + 19f_1g_{14} + 19f_2g_{13} + 19f_3g_{12} + 19f_4g_{11} + 19f_5g_{10} + 19f_6g_9 + 19f_7g_8 + 19f_8g_7 + 19f_9g_6 + 19f_{10}g_5 + 19f_{11}g_4 + 19f_{12}g_3 + 19f_{13}g_2 + 19f_{14}g_1\\
h_1 &= f_0g_1 + f_1g_0 + 19f_2g_{14} + 19f_3g_{13} + 19f_4g_{12} + 19f_5g_{11} + 19f_6g_{10} + 19f_7g_9 + 19f_8g_8 + 19f_9g_7 + 19f_{10}g_6 + 19f_{11}g_5 + 19f_{12}g_4 + 19f_{13}g_3 + 19f_{14}g_2\\
h_2 &= f_0g_2 + f_1g_1 + f_2g_0 + 19f_3g_{14} + 19f_4g_{13} + 19f_5g_{12} + 19f_6g_{11} + 19f_7g_{10} + 19f_8g_9 + 19f_9g_8 + 19f_{10}g_7 + 19f_{11}g_6 + 19f_{12}g_5 + 19f_{13}g_4 + 19f_{14}g_3\\
h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + 19f_4g_{14} + 19f_5g_{13} + 19f_6g_{12} + 19f_7g_{11} + 19f_8g_{10} + 19f_9g_9 + 19f_{10}g_8 + 19f_{11}g_7 + 19f_{12}g_6 + 19f_{13}g_5 + 19f_{14}g_4\\
h_4 &= f_0g_4 + f_1g_3 + f_2g_2 + f_3g_1 + f_4g_0 + 19f_5g_{14} + 19f_6g_{13} + 19f_7g_{12} + 19f_8g_{11} + 19f_9g_{10} + 19f_{10}g_9 + 19f_{11}g_8 + 19f_{12}g_7 + 19f_{13}g_6 + 19f_{14}g_5\\
h_5 &= f_0g_5 + f_1g_4 + f_2g_3 + f_3g_2 + f_4g_1 + f_5g_0 + 19f_6g_{14} + 19f_7g_{13} + 19f_8g_{12} + 19f_9g_{11} + 19f_{10}g_{10} + 19f_{11}g_9 + 19f_{12}g_8 + 19f_{13}g_7 + 19f_{14}g_6\\
h_6 &= f_0g_6 + f_1g_5 + f_2g_4 + f_3g_3 + f_4g_2 + f_5g_1 + f_6g_0 + 19f_7g_{14} + 19f_8g_{13} + 19f_9g_{12} + 19f_{10}g_{11} + 19f_{11}g_{10} + 19f_{12}g_9 + 19f_{13}g_8 + 19f_{14}g_7\\
h_7 &= f_0g_7 + f_1g_6 + f_2g_5 + f_3g_4 + f_4g_3 + f_5g_2 + f_6g_1 + f_7g_0 + 19f_8g_{14} + 19f_9g_{13} + 19f_{10}g_{12} + 19f_{11}g_{11} + 19f_{12}g_{10} + 19f_{13}g_9 + 19f_{14}g_8\\
h_8 &= f_0g_8 + f_1g_7 + f_2g_6 + f_3g_5 + f_4g_4 + f_5g_3 + f_6g_2 + f_7g_1 + f_8g_0 + 19f_9g_{14} + 19f_{10}g_{13} + 19f_{11}g_{12} + 19f_{12}g_{11} + 19f_{13}g_{10} + 19f_{14}g_9\\
h_9 &= f_0g_9 + f_1g_8 + f_2g_7 + f_3g_6 + f_4g_5 + f_5g_4 + f_6g_3 + f_7g_2 + f_8g_1 + f_9g_0 + 19f_{10}g_{14} + 19f_{11}g_{13} + 19f_{12}g_{12} + 19f_{13}g_{11} + 19f_{14}g_{10}\\
h_{10} &= f_0g_{10} + f_1g_9 + f_2g_8 + f_3g_7 + f_4g_6 + f_5g_5 + f_6g_4 + f_7g_3 + f_8g_2 + f_9g_1 + f_{10}g_0 + 19f_{11}g_{14} + 19f_{12}g_{13} + 19f_{13}g_{12} + 19f_{14}g_{11}\\
h_{11} &= f_0g_{11} + f_1g_{10} + f_2g_9 + f_3g_8 + f_4g_7 + f_5g_6 + f_6g_5 + f_7g_4 + f_8g_3 + f_9g_2 + f_{10}g_1 + f_{11}g_0 + 19f_{12}g_{14} + 19f_{13}g_{13} + 19f_{14}g_{12}\\
h_{12} &= f_0g_{12} + f_1g_{11} + f_2g_{10} + f_3g_9 + f_4g_8 + f_5g_7 + f_6g_6 + f_7g_5 + f_8g_4 + f_9g_3 + f_{10}g_2 + f_{11}g_1 + f_{12}g_0 + 19f_{13}g_{14} + 19f_{14}g_{13}\\
h_{13} &= f_0g_{13} + f_1g_{12} + f_2g_{11} + f_3g_{10} + f_4g_9 + f_5g_8 + f_6g_7 + f_7g_6 + f_8g_5 + f_9g_4 + f_{10}g_3 + f_{11}g_2 + f_{12}g_1 + f_{13}g_0 + 19f_{14}g_{14}\\
h_{14} &= f_0g_{14} + f_1g_{13} + f_2g_{12} + f_3g_{11} + f_4g_{10} + f_5g_9 + f_6g_8 + f_7g_7 + f_8g_6 + f_9g_5 + f_{10}g_4 + f_{11}g_3 + f_{12}g_2 + f_{13}g_1 + f_{14}g_0
\end{aligned}
$$

Fig. 2. Multiplication of numbers f and g in the radix-2^{17} representation

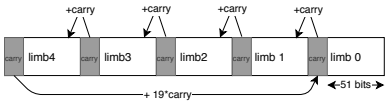

Fig. 3. reduce operation for radix-2^{51} representation.

`-O3` flag. From this, we concluded that the compiler does the best possible inlining when the code is compiled with `-O3` and no further manual inlining was required.

Eliminating the *reduce* function. As discussed in section 2 performing field arithmetic limb-wise can result in an overflow of the limbs. For example, in the radix-2^{17} representation, adding two 17 bit limbs can result in an 18 bit limb. This overflow must then be carried over to the next limb. If the last limb overflows, the number needs to be reduced modulo $2^{255} - 19$ to be a field element again. This is the *reduce* operation, Fig. 3 illustrates this operation for the radix-2^{51} implementation.

The computations for every limb depend on the previous limb, giving the whole computation a cyclic structure, and so impossible to parallelize and hence cannot be vectorized. We presume that this means that *reduce* will be a bottleneck. In the *Curve25519* computations, we can make use of the fact that the result of *add* is always used as an operand in *mult* or *sqr*. When this is the case, the reduction step is no longer necessary after every *add* [2] as it suffices that this is done in *mult* or *sqr* respectively. For *sub* the result only needs to be reduced if any of the limbs are negative.

Vectorization using Intel Intrinsics. The curve computations cannot be vectorized due to the interdependencies between the computation steps in the scalar multiplication. We therefore focused on vectorizing the big number operations.

Loading, storing and permuting the limbs of vectors can produce a significant overhead and therefore these operations must be used with caution. Due to the complex computation involved in the operations *sqr* and *mult* (see Fig. 2), reducing the number of permutations was the main challenge.

To avoid the overhead of loading from and storing to memory, we replaced the 15-limb radix-2^{17} representation with two variables of vector type `__m256i`. We then changed the type of all arguments of functions to these vectors.

The vectorization of the operations *add* and *sub* was straight-forward, as we replaced the addition and subtraction of each limb with two vector additions and two vector subtractions respectively. What proved to be trickier was the conditional use of the *reduce* function in the *sub* operation. This was done using a combination of the intrinsic function `_mm256_movemask_ps`. While somewhat costly, this still performed better than calling *reduce* when unnecessary.

As mentioned above, the vectorization of the *sqr* and *mult* operations were less straight-forward by their nature. As seen in Fig. 2, the multiplication has a double triangular shape, which necessitates permutations, making a simple vector addition or multiplication impossible.

In the following paragraphs we outline a technique we devised to minimize the number of permutations for the *mult* operation. A similar technique was used for *sqr*.

The main feat was the restructuring of the computation. In a first step, we split each equation in two, introducing a larger result vector, as proposed in [7]. All multiplication terms in h_i (highlighted in Fig. 2) are moved to h'_{i+15}, with h'_i containing the remaining terms. For all i in $\{0..13\}$ it then holds that $h_i = h'_i + 19 * h'_{i+15}$ and $h_{14} = h'_{14}$ (see also [7]). The additional step of calculating h_i from h'_i is done at the very end.

At first glance, this does not reduce the number of permutations. However, after some additional restructuring of the multiplication terms, a convenient shape emerges (see Fig. 4). This shape allows us to minimize the number of

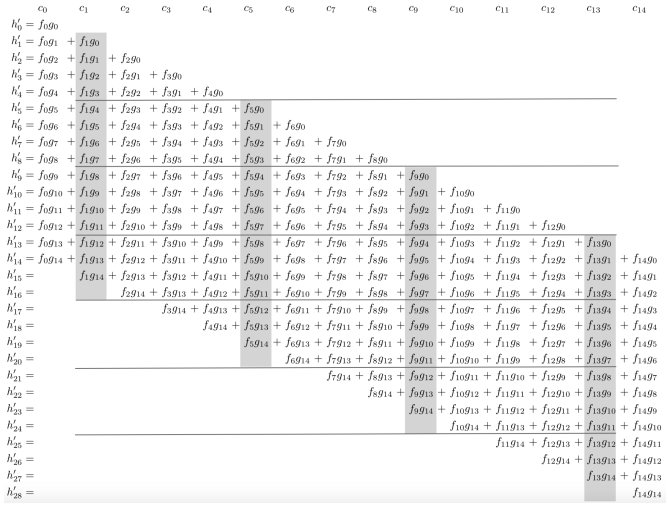

Fig. 4. Restructured multiplication of numbers f and g in the radix-2^{17} representation with the result vector of each column $c_i = f_i * \mathbf{g}$. The highlighted areas show an example of how columns may be grouped and which 4-way SIMD vectors are added together (the case shown is where the offset from the starting point modulo 4 equals 1).

permutations.

As we can see in Fig. 4, the multiplication terms can be grouped into so-called columns. In each column we multiply the simple input vectors of g with a vector consisting of a single element of the operand f. It therefore suffices to prepare the necessary vectors only once at the start of the function and then reuse them to compute the result of each column.

The complexity is thus reduced to the addition of the columns. As visible in Fig. 4, the starting position of the vector elements move down by one index for every column, making a straight-forward addition impossible. Instead of permuting the result vectors of each column at this point, we group the columns into the ones which can be added without permutations, namely the ones for which the offset from the starting position modulo 4 is the same[7].

By doing this we are left with only four columns, for whose addition permutations are necessary. Thus we obtain the h'_i and can from there compute the h_i as described above.

Optimizing curve computations. By mathematically restructuring the curve computations, `point_add` and `point_double`, we significantly reduced the number of the costly 255-bit arithmetic operations (*div*, *mult* and *sqr*) required for scalar multiplication. We achieved this by:

- pre-computing and reusing results of costly operations
- performing repetitive additions instead of multiplication by small constants
- extracting common factors in order to reduce the number of *mult*.

As shown in Table 2, applying the aforementioned optimizations leads to a lower number of calls to arithmetic functions and results in a significant speedup.

		add	sub	mult	sqr	div
double	base	9	5	11	4	2
double	opt	9	2	3	2	1
add	base	3	9	5	4	3
add	opt	4	4	2	1	1

Table 2. Number of 255-bit operations per curve computations comparing the base to the optimized version

Microbenchmarking. Initial performance measurements on our optimizations did not yield the expected speedup. We therefore decided to inspect the runtime of each big number operation (i.e. *add*, *mult*, etc.), and any relevant optimizations thereof, individually by using a microbenchmarking framework.

As we suspected the *reduce* function to be a bottleneck (see previous paragraph), we inspected the runtime of the operations *add* and *sub*, without – or in the case of *sub* only conditionally – executing the *reduce*.

The microbenchmarking results enabled us to make informed decisions regarding the choice of the big number operation and its exact implementation for use in the curve computations. For example, the microbenchmarking confirmed that the *sqr* operation was in all cases faster than the *mult* and it would therefore be advantageous to calculate a^2 using the former.

The microbenchmarking framework was often used in an iterative process during development to ensure certain adjustments gave us the desired results.

Combining all optimizations. We then combined the individual optimization techniques described in the previous paragraphs and used the insights from the microbenchmarking to form a separate optimization which resulted in a significant gain in performance.

Optimizing radix-2^{51} and radix-$2^{25.5}$ implementations. Using techniques devised for the radix-2^{17} optimizations (scalar replacement, curve optimizations and vectorization), we performed similar optimizations on the base implementations of other radix representations.

[7]In the *mult* operation, we operate on 64-bit and not 32-bit integers, which means that the additions are 4-way SIMD instructions. The result vector of each column c_i is in reality 4 `__m256i` vectors of 64-bit integers. When adding columns whose offset modulo 4 is the same, we can then simply add the relevant vectors.

While the techniques could be almost directly translated to the radix-$2^{25.5}$ representation, this was not the case for radix-2^{51}. The reason behind this is that it is not possible to fully vectorize radix-2^{51}. A full vectorization would require an AVX2[8] Intrinsic instruction to multiply 64-bit integers, which does not exist [8]. We therefore vectorized the parts which were possible, which resulted in a slight overhead due to the load to and store from the vector type `__m256i`.

No optimizations for cache misses. The inputs to the public key generation algorithm are a fixed base point and the secret key (see section 2) which is read bit-wise in a loop. All inputs to the functions are read only once, and so cache misses are not a concern. We attempted to artificially introduce cache misses by performing scalar multiplications for many secret keys in parallel. We performed these computations for up to 2^{13} secret keys simultaneously but observed no degradation in performance. We verified the execution with the tool *Cachegrind* [9] and confirmed that there were no cache misses.

We therefore concluded that memory hierarchy optimizations are not relevant to this project.

4. EXPERIMENTAL RESULTS

All our experiments were run on a single machine using a benchmarking and microbenchmarking framework. The former also included a validation functionality, which we ran every time to ensure that any optimization still yielded a correct result.

In the following paragraphs we give an overview of the validation and benchmarking process and the machine setup before presenting our final results.

Validation. Our validation consists of two parts:

1. Verification of the generated public key
2. Simulating a full Diffie-Hellman key exchange and verifying that the shared secret established in the end is the same for both parties

For the first part, we used our base implementation using the GMP library to create the ground truths. In order to ensure the correctness of that implementation, we also simulated a full Diffie-Hellman key exchange and additionally cross-checked a sample of our results with a third-party library for *Curve25519* key generation [10].

While, for our final results, we only generated the public key for one secret key[9], we used larger test sets (with up to 1024 keys) for the validation of our implementations. This gave us a high degree of certainty of the correctness of all our implementations.

Benchmarking setup. We set up a benchmarking framework, written in C and complemented by a bash script. The bash script `run_benchmarking.sh` which can be found in our git repository can be used to select the implementations to be benchmarked, as well as the compiler flags to be used.

Before running the benchmarking process, the caller initializes any required big number constants (e.g. the curve parameters, SIMD[10] vectors used as masks etc.).

The function to be benchmarked is the scalar multiplication, which takes as input the base point and the secret key provided by the test set.

After a warm-up phase we ran the scalar multiplication multiple times and returned the average number of cycles for a more statistically relevant result.

Microbenchmarking setup. The microbenchmarking framework was set up separately. It creates a random test set and then runs each operation (*add*, etc.) to be benchmarked multiple times (after an initial warm-up phase) on each test case. The framework measures exactly the number of cycles required to run the function with no additional overhead. It returns the average number of cycles over the number of runs and test cases.

Machine setup. All experiments were run on an Intel(R) Core(TM) i7-6600U CPU. Intel Turbo Boost was disabled to keep the CPU at a frequency of 2.60GHz. The machine runs the operating system Ubuntu 18.04.2.

Compiler settings. All code was compiled using *gcc* (version 7.4.0). We chose four different sets of compiler flags to focus on. We refer to the names provided in italic below as *compiler settings* in the following discussion and plots.

1. *no-opt*: no compiler flags used
2. *o2*: `-O2 -mavx2 -march=native -mfma`
3. *o3-novect*: `-O3 -march=native -mfma -fno--tree-vectorize`
4. *o3*: `-O3 -mavx2 -march=native -mfma`

Results of radix-2^{17} optimizations. As we can see in Fig. 5, performing scalar replacement as discussed in section 3 results in very little performance gain, depending on

[8] Advanced Vector Extensions 2

	base	all opt	speedup
radix-2^{51}	0.67	2.13	3.18
radix-$2^{25.5}$	1.27	3.81	3.01
radix-2^{17}	1.05	4.04	3.84

Table 3. Comparing the performance ([iops/cycle]) of the base implementations with the all optimized versions for the various radix representations (compiler setting *o3*).

[9] As mentioned in section 3, the size of the input has no effect on the performance, which is why it is sufficient to run the experiments on a single test case

[10] Single instruction, multiple data

the compiler setting even a performance decrease. Vectorization performed using the techniques discussed in section 3, improves the performance, but does not give us the expected 4x speedup[11]. This is most likely due to a significant computational overhead from the permutations required for *mult* and *sqr* computations. It is interesting to note that, when we do not use any Intrinsic functions inside the implementation, like in the optimizations of the curve computations, the performance decreases when the compiler tries to vectorize (compare the performance of *curve opt* with settings *o3-novect* and *o3* in Fig. 5). When we perform vectorization, as discussed in section 3, the performance with compiler settings *o3-novect* and *o3* are essentially the same. This indicates that there is no more space left for the compiler to vectorize further (compare the performance of *scalar repl. & vect.* and *all opt* with settings *o3-novect* and *o3* in Fig. 5). We conclude from this result that we have probably devised the most ideal technique in section 3 to vectorize.

Combining all optimizations as discussed in section 3 into one implementation gives us a speedup of 3.84x (see Table 3). Fig. 6 shows the roofline plot for the machine and plots performance for the different optimizations of the radix-2^{17} implementation. We note that *all opt* just slightly crosses the scalar roofline.

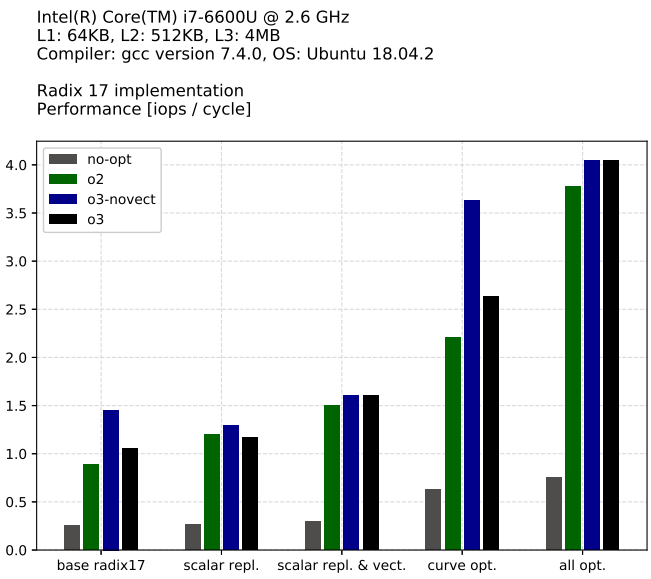

Fig. 5. Performance of different optimizations (radix-2^{17})

Microbenchmarking results. We provide here the most interesting results from the microbenchmarking and the conclusions we draw from them:

[11] In the operations *mult* and *sqr* we use 4-way SIMD instructions since we perform 32-bit multiplications (which result in 64-bit results) and 64-bit additions. Since these operations are the ones we use the most, we would expect ≈ 4x speedup.

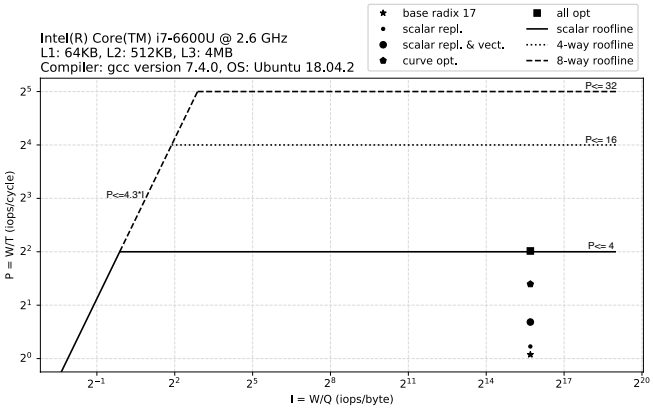

Fig. 6. Roofline plot for different optimizations for the radix-2^{17} implementation

1. ***Reduce* function as a bottleneck**: As expected the reduce function proved to be a bottleneck. In the radix-2^{17} base implementation we achieved a 3.78x speedup (without compiler optimizations) when removing the call to the *reduce* function in the *add* operation.
   When the *reduce* is called the speedup of the vectorized version compared to the base implementation is only 5.95x. However, on removing the *reduce* in both implementations we achieved a 13.81x speedup. We assume the massive speedup in the vectorized version of *add* without calling *reduce* is due to the fact that the compiler can take advantage of pipelining the now fully independent operations.
   Since *reduce* does not have to be called after an *add* and only conditionally after a *sub* operation (see section 3), the fact that we can achieve such a speedup without it is very useful. However, *reduce* remains a bottleneck where it is mandatory, i.e. in *sqr* and *mult*.
2. **Vectorization of *mult* and *sqr***: Unfortunately, the speedup obtained by vectorizing the radix-2^{17} implementation of the *mult* operation was minimal, 1.67x compared to the radix-2^{17} base implementation (using no compiler flags). The results were even worse for the *sqr* operation where we could only achieve a 1.12x speedup. As mentioned in section 3 this is most likely due to the overhead caused by the unavoidable but costly permutations, and as mentioned previously the *reduce* function.
   The situation changes however, when it is compiled using the *o3-novect* compiler setting. For the *mult* operation we now achieved a 5.43x speedup compared to the base implementation and for *sqr* a marginally better 1.57x speedup. We assume here that by using

the technique described in section 3 to vectorize *mult*, we have optimized the structure of the computation in such a way that it allows the compiler to apply further optimizations it could not perform on the non-ideal structure in the base implementation. In particular, we have introduced many fully independent computations, giving the compiler more room for scheduling and pipelining optimizations.
This result reinforces the conclusions we draw in the previous paragraph.

3. ***Inv* operation as a bottleneck**: With 254 *sqr* and 11 *mult* operations used (see section 2), the *inv* is by far the most costly operation [12]. In fact *inv* takes 162.5x longer than the next most costly function (*mult*). Reducing the amount of big number divisions as done in our curve optimizations (see section 3) was therefore a crucial factor in the speedup obtained by the curve optimizations.

Comparing results from different radix implementations. To compare the performance of the different radix representations we used the second cost measure, as introduced in section 2. Since the performance is then inversely proportional to the runtime, we directly used the runtime for the comparison.

Fig. 7 shows the runtime of the base implementations and the implementations with all optimizations from section 3 combined for the different radix representations. Additionally, it plots the performance of the implementation using GMP. Our optimizations for all radix representations achieve a significant decrease in runtime compared to the base implementations of the respective radix. Interestingly, the lowest runtime is achieved by the radix-2^{51} implementation, in spite of the fact that some arithmetic functions cannot be fully vectorized using AVX2. This can be explained by the fact that the number of low-level operations grows polynomially in the number of limbs of the representation as illustrated in section 2.

5. CONCLUSIONS

We focused our optimizations on the radix-2^{17} implementation. As seen in Table 3, the obtained speedup for radix-2^{17} is the highest compared to the other radix representations. This means that our hypothesis stated in section 2 holds.

However, if our implementation would be used for a real-world public key generation in ECDHE, the runtime would be a more critical measure than the performance in iops/cycle. Fig. 7 shows that the optimized version of radix-2^{51} has the lowest runtime and thus would be the most useful. Existing libraries that use radix-$2^{25.5}$ representation like [10], run faster than our implementations. However,

[12]except for the *div* operation, which is a simple *inv* and *mult*

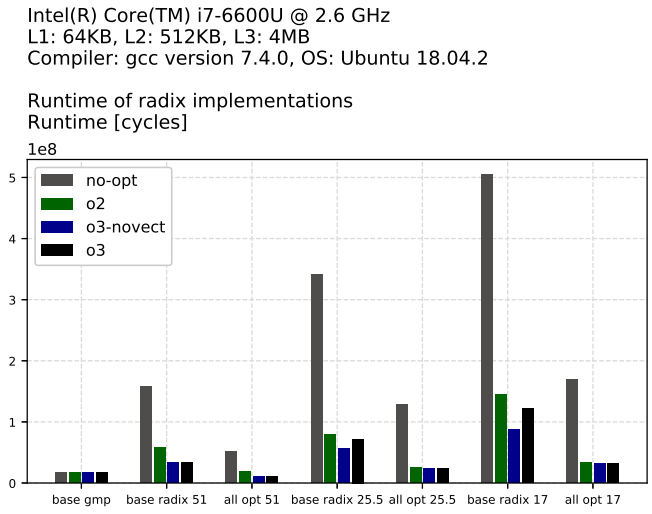

Fig. 7. Runtime comparison of different radix implementations

they perform further optimizations on the Assembly code (which was out of the scope of this project). This could be the reason for the faster runtime.

6. FUTURE WORK

We did not analyze if our optimizations compromised the side channel resilience given by the Montgomery ladder. In particular, it would be interesting to investigate if the conditional *reduce* in *sub* opens up a timing side channel leading to information leakage.

This project mainly focused on optimizations that were possible directly in the C code. However, [4] explains different techniques that can be used to further optimize the Assembly code. Optimizing the Assembly for the radix-2^{17} implementation and comparing this with the existing libraries that use the radix-$2^{25.5}$ could be an extension of this work.

7. CONTRIBUTIONS OF TEAM MEMBERS

This project was very much a collaborative work with every member of the team participating equally in all the optimizations. In particular, we all contributed to the debugging of all implementations and complemented each others' code whenever necessary. Below we list the most significant contribution by each member.

Supraja. Implemented the base implementations together with Robin and wrote scripts for conversions from decimal to the different radix representations and vice versa. Analysed the code for possibility of memory optimizations and restructured the computations to reinforce the conclu-

sion that memory optimizations were not relevant for this project. Performed function inlining as explained in section 3 and analysed the results. Instrumented the different radix representations to count the number of operations for both cost measures. Worked on vectorizations of some big number arithmethic functions together with Robin.

Manuel. Worked together with Robin on the GMP implementation. Focused on the optimizations of the curve computations and the vectorization of big number arithmetic in the different radix representations. Ensuring validation of all optimizations through extensive debugging.

Robin. Implemented the base GMP version together with Manuel, and the radix base implementations with Supraja. Worked on scalar replacement and vectorization of big number arithmetic in the different radix representations, helped defining curve optimizations, ran benchmarks and generated plots with Supraja. In particular implemented the vectorization of *sqr* and *mult* as described in section 3 for radix-2^{17} and radix-$2^{25.5}$ together with Sabina and Manuel.

Sabina. Implemented the benchmarking and microbenchmarking framework. Focused on vectorization of big number arithmetic in the different radix representations, helped defining curve optimizations. Contributed to scalar replacement. In particular, implemented the vectorization of *sqr* and *mult* as described in section 3 for radix-2^{17} and radix-$2^{25.5}$ together with Robin and Manuel. Combined all optimizations for the radix-2^{51} representation. Incorporated results obtained from microbenchmarking.

8. REFERENCES

[1] Daniel J. Bernstein and Tanja Lange, "Safe curves: choosing safe curves for elliptic-curve cryptography," .

[2] Daniel J. Bernstein, "Curve25519: New diffie-hellman speed records," in *Public Key Cryptography - PKC 2006*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, Eds., Berlin, Heidelberg, 2006, pp. 207–228, Springer Berlin Heidelberg.

[3] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 7748, RFC Editor, August 2018.

[4] Tung Chou, "Sandy2x: New curve25519 speed records," 01 2016, vol. 9566, pp. 145–160.

[5] Daniel J. Bernstein and Tanja Lange, *Montgomery Curves and the Montgomery Ladder*, p. 82–115, Cambridge University Press, 2017.

[6] M. Hamburg A. Langley and S. Turner, "Elliptic Curves for Security," RFC 7748, RFC Editor, January 2016.

[7] Peter Schwabe, "Finite field arithmetic," University Lecture, 2013.

[8] Intel Corporation, "Intrinsics guide," (year not specified).

[9] Valgrind Developers, "Cachegrind," 2020.

[10] Free Software Foundation, "Reference curve255lib," 2013.

9. ACRONYMS

AVX2 Advanced Vector Extensions 2

ECC Elliptic Curve Cryptography

ECDHE Elliptic Curve Diffie-Hellman Key Exchange

GF Galois-Field

GMP GNU Multiple Precision Arithmetic Library

RFC Request For Comments

SIMD Single instruction, multiple data

TLS Transport Layer Security