Parallelized Montgomery Exponentiation in GF$\left(2^k\right)$ for Diffie–Hellman Key Exchange Protocol

Alexander Krikun and Alla Levina

***Abstract*—Finite field arithmetic is commonly used in cryptographic applications such as establishing secure connections between systems. Recent standards have expanded the necessary key lengths, potentially increasing the time connections might take. This paper offers performance tests of highly efficient optimization structure for modular exponentiation based on parallelizing Montgomery exponentiation in GF$\left(2^k\right)$. The paper also describes the algorithms necessary for an implementation of the optimization structure in question and performance tests for a Diffie–Hellman Key Exchange system utilizing the structure. The resulting efficiency improvements of nearly 45% over standard modular exponentiation can potentially be applied to any algorithm that relies on such operations.**

***Index Terms*—Cryptography; Diffie-Hellman Key Exchange; Montgomery exponentiation**

I. Introduction

MODULAR multiplication and exponentiation are fundamental arithmetic operations underlying most of the currently deployed public-key cryptographic protocols [1]. The cryptographic security of such protocols is derived from one-way functions - functions that is easy to compute (in polynomial time) on a random input, but infeasible to invert (only possible in non-polynomial time) given an image of a random input. For instance, the Diffie–Hellman key exchange is based on discrete exponentiation and reduction modulo p. A modular exponent can be calculated in polynomial time, but inverting it requires a solution to the discrete logarithm problem.

The original implementation of the algorithm uses multiplicative groups of integers modulo p, where p is a prime. However, contemporary implementations are defined over a Galois field GF$\left(2^k\right)$ with an irreducible polynomial $n(x)$. Polynomial representations of the elements of this field are particularly suitable for both software and hardware implementation, and arithmetic operations in GF$\left(2^k\right)$ are widely applied in programming, computer algebra and cryptography. Furthermore, finite field Diffie-Hellman is one of the two types still in use as of transport layer security protocol TLS 1.3 [2] (the other being Elliptic curve Diffie–Hellman – ECDH), and operations in GF(2^k) are implemented in OpenSSL [3].

Multiplication in GF$\left(2^k\right)$ is multiplication modulo an irreducible polynomial used to construct that finite field. The irreducible polynomial can have several relevant properties. For example, the polynomials that define Oakley groups 1 and 2 [4] were chosen to be Sophie Germain primes for maximum resistance against the square-root attack on the discrete logarithm problem. The irreducible polynomials to be used for the Diffie–Hellman Key Exchange (DHKE) require rigorous testing to ensure they are sufficiently strong and verify the discrete logarithm problem is indeed difficult. It is considered a best practice to use pre-defined irreducible polynomials provided by the Internet Engineering Task Force (IETF) [4], [5], [6]

The Diffie–Hellman algorithm requires an implementation of the exponentiation g^E, where g is a fixed primitive element of the field and E is an integer. This exponentiation operation can be implemented using a series of squaring and multiplication operations in GF$\left(2^k\right)$ using the binary method [7]. Lookup tables can be used to speed up multiplications at the expense of memory, but they quickly become impractical with larger orders k. With large values of k being desirable for cryptographic operations, faster algorithms are essential.

One such algorithm is Montgomery exponentiation [8], which in its standard implementation already shows a 20% to 40% improvement with key lengths of over 1024 bits. A parallelized implementation of the Montgomery exponentiation algorithm [9], however, shows an even more significant improvement of up to 50%. This optimization enables cryptographic systems to use longer keys while utilizing the same computational time, which improves system security.

This paper presents an implementation of the parallelized Montgomery exponentiation algorithm along with comparative performance testing. The Diffie–Hellman algorithm will be focused on as a possible optimization target, but the performance gains are applicable to any algorithm that relies on modular exponentiation.

II. Regulations and issues

Cryptography is subject to a wide array of regulations. The United States severely limited the export of cryptographic technology and devices until 2000, and export-grade cryptography must have been no stronger than 40 bits. Due to asymmetric cryptography requiring longer keys for the equivalent level of security this meant export-grade Diffie–Hellman could utilize keys no longer than 512 bits. These limits have long been lifted, but parameters for export-grade cryptography were still supported on nearly 10% of the top 1 million domains as of 2015 [10]. 512-bit keys are dangerously short, close to the 400-bit theoretical limit for the discrete logarithm problem established in 1984 [11]. The widespread support for these parameters led to attacks such as Logjam [10], FREAK [12] and DROWN [13]. Logjam was an attack on Diffie-Hellman exploiting both downgrade attacks and novel methods of number sieve field precomputation. The pre-computation stage for 512-bit primes

Manuscript received December 03, 2020; revised February 9, 2021.

Alexander Krikun is a MD. student, School of Translational Information Technologies, ITMO University, Saint-Petersburg 197101, Russia, krikun98@gmail.com

Alla Levina is an Associate Professor, Saint Petersburg Electrotechnical University "LETI", 197376, Saint-Petersburg, Professora Popova str., 5, Russia, alla_levina@mail.ru.

took less than a week, and the individual discrete logarithm time was 70 seconds. Export-grade cryptography has largely been deprecated, but active regulations still allow relatively weak keys.

After the recent deprecation of TLS 1.0 and 1.1 implementations of the Diffie–Hellman key exchange protocol can use parameters provided by TLS 1.2 onwards. TLS 1.2, however, still allows the use of 768-bit keys derived with the first OAKLEY group. The latest IETF recommendations suggest never using these. Keys derived with this group can, quote, "be broken within hours using cheap off-the-shelf hardware" and "provide no security whatsoever" [14]. The smallest keys recommended for finite field Diffie Hellman are at least 2048-bit [5]. TLS 1.3 has also made it a requirement to use ephemeral keys, i.e., to generate new keys for each key exchange.

With the latest developments and requirements in place the speed of the key exchange becomes rather important. The length of the minimal keys considered secure may grow two to four times, and caching secret keys for several operations is now forbidden. The optimization structure proposed in this paper accelerates modular exponentiation, reducing the overall duration of the key exchange.

Another option would be to use elliptic curve cryptography, which provides the same level of security with shorter keys. Elliptic Curve Diffie–Hellman is the second option provided by TLS 1.3. Research on Montgomery-like representations of elliptic curves has shown inspiring results [15], and similar optimizations might be applicable there.

III. Polynomial representation

The polynomial representation of elements of the field $\mathrm{GF}(2^k)$ is especially suitable for software implementation [8]. The algorithms for Montgomery exponentiation in this paper are based on this polynomial representation. An element a of $\mathrm{GF}(2^k)$ is represented as a polynomial of length k, i.e., of a degree less than or equal to $k-1$, written as

$$a\,(x) = \sum_{i=0}^{k-1} a_i x^i = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \ldots + a_1 x + a_0$$

where the coefficients $a_i \in GF^2$. These coefficients are also referred to as the bits of a, and the element a is represented as $a = (a_{k-1}, a_{k-2}, \ldots a_1, a_0)$. Specific bits can also be addressed as a_i.

Further explanations of exponentiation algorithms will feature numerical examples with the 8-bit irreducible polynomial $10001001(137)$. Both practical implementations and further tests use polynomials thousands of bits long, but that would be infeasible for demonstrational purposes.

The addition of two elements a and b in $\mathrm{GF}(2^k)$ is performed by adding the polynomials $a(x)$ and $b(x)$, where the coefficients are added in the field GF(2). This is equivalent to the bitwise XOR operation on the vectors a and b.

$$\begin{array}{rrr} + & 63 & 111111 \\ & 42 & 101010 \\ \hline & 21 & 10101 \end{array}$$

Division by x represents a bitwise shift one bit to the right with the trailing bit cut off. Unlike ordinary division, this is a highly efficient operation. Simularly, multiplication by x represents a bitwise left shift.

$$\begin{array}{rrr} / & 63 & 111111 \\ & x & 1 \\ \hline & 31 & 11111 \end{array}$$

Multiplication is performed with respect to an irreducible polynomial of degree k. Let $n(x)$ be an irreducible polynomial of degree k over the field $\mathrm{GF}(2^k)$. The product $c = a \cdot b$ in $\mathrm{GF}(2^k)$ is then calculated as

$$c(x) = a(x) \cdot b(x) \pmod{n(x)}$$

where $c(x)$ is a polynomial of length k, representing the element $c \in GF(2^k)$. The bit-level algorithm for multiplication is based on a modification of the peasant algorithm:

Input: $a(x)$, $b(x), n(x)$
Output: $c(x) = a(x) \cdot b(x) \pmod{n(x)}$
1: $c(x) = 0$
2: $n_{mult}(x) = n(x) - n_{k-1}x^{k-1}$
3: **for** i=0 **to** k−1
4: $\quad c(x) = c(x) + b_0 a(x)$
5: $\quad b(x) = b(x)/x$
6: $\quad car = a_{k-1}$
7: $\quad a(x) = a(x) \cdot x$
8: $\quad$ **if** $car == 1$: $a(x) = a(x) - n_{mult}(x)$

$n_{mult}(x)$ is the irreducible polynomial with the high term eliminated. *car* represents the carry - the high term of $a(x)$ that gets eliminated in step 7. Conceptually, $a(x) \cdot x - n_{mult}(x) \equiv a(x) \cdot x \pmod{n(x)}$ if the high term of $a(x)$ was 1.

$$\begin{array}{rrrr} \times & 63 & & 111111 \\ & 3 & & 11 \\ \hline & 65 \pmod{137} & 1000001 & \pmod{10001001} \end{array}$$

The standard left-to-right binary exponentiation algorithm is as follows [16]:

Input: $m(x)$, e, $n(x)$
Output: $c(x) = m(x)^e \pmod{n(x)}$
1: $c(x) = 1$
2: **for** i=k−1 **downto** 0
3: $\quad c(x) = c(x) \cdot c(x)$
4: $\quad$ **if** $e_i == 1$: $c(x) = m(x) \cdot c(x)$

$$\begin{array}{rrrr} \hat{} & 63 & & 111111 \\ & 2 & & 10 \\ \hline & 15 \pmod{137} & 1111 & \pmod{10001001} \end{array}$$

This algorithm is not the only option for binary exponentiation, and alternatives were implemented for the tests in this paper. They, however, all displayed similar results, so this algorithm was chosen as the performance baseline.

IV. Montgomery exponentiation

Montgomery representation allows efficient implementations of modular multiplications without explicitly carrying out the classical modular reduction step.

Let n be a positive integer, and let r and a be integers such that $r > n$, $\gcd(n, r) = 1$ and $0 \le a < nR$.

The Montgomery form of a is calculated as follows:

$$\overline{a} = a \cdot r \pmod{n}$$

The advantage of the Montgomery form is that most of the arithmetic mod n is instead computed in mod r by

using Montgomery multiplications. If r is selected to be a factor of two, these operations will be computed much faster, because a bitwise right shift will replace the costly division and a bit mask — a modulo. Bit-level algorithms make the implementational benefit even greater by only relying on two operations – bitwise XOR and single-bit right shift, both inherently fast operations – for a Montgomery multiplication. Montgomery multiplication is also applicable to calculations in GF(2^k).

Bit-level algorithm for Montgomery multiplication (MMM) [16]:

Input: $\overline{a}(x)$, $\overline{b}(x), n(x)$
Output: $\overline{c}(x) = c(x) \cdot r(x) \pmod{n(x)}$
1: $\overline{c}(x) = 0$
2: for i=0 to k−1
3: $\quad \overline{c}(x) = \overline{c}(x) + \overline{a}_i \overline{b}(x)$
4: $\quad \overline{c}(x) = \overline{c}(x) + \overline{c}_0 n(x)$
5: $\quad \overline{c}(x) = \overline{c}(x)/x$

Montgomery multiplications are performed on Montgomery forms of polynomials:
$\overline{c} \equiv \overline{a} \cdot \overline{b} \cdot r^{-1} \pmod{n} \equiv a \cdot r \cdot b \cdot r \cdot r^{-1} \pmod{n} \equiv c \cdot r \pmod{n}$

Therefore, we first have to convert both factors to the Montgomery form, and re-convert them back after via a Montgomery multiplication by 1 (which is effectively a multiplication by r^{-1}, removing r).

$\overline{63} = 92, \overline{3} = 27$

MMM		$\overline{63}$		1011100
		27		11011
MMM	109 (mod 137)	1101101	(mod 10001001)	
		1		1
	65 (mod 137)	1000001	(mod 10001001)	

This adds a slight computational overhead on the conversion to the Montgomery form and back. However, when performing Montgomery multiplications in sequence, like in the exponentiation operation, the speed advantage of the Montgomery form outweighs it.

Bit-level algorithm for Montgomery exponentiation [8]:

Input: $m(x)$, e, $n(x)$, $R_2 = r(x) \cdot r(x) \pmod{n(x)}$
Output: $c(x) = m(x)^e \pmod{n(x)}$
1: $\overline{m}(x) = MMM(m(x), R_2)$
2: $\overline{c}(x) = MMM(1, R_2)$
3: for i=0 to k−1
4: $\quad$ if e_i == 1: $\overline{c}(x) = MMM(\overline{m}(x), \overline{c}(x))$
5: $\quad \overline{m}(x) = MMM(\overline{m}(x), \overline{m}(x))$
6: $c(x) = MMM(\overline{c}(x), 1)$

^		$\overline{63}$		1011100
		2		10
MMM	119 (mod 137)	1110111	(mod 10001001)	
		1		1
	15 (mod 137)	1111	(mod 10001001)	

Montgomery multiplications can be improved and accelerated in several ways [1], and specialized hardware exists for such purposes [13]. Montgomery exponentiation, however, can be optimized further.

V. Parallel Montgomery Exponentiation

As explained in [9], during a Montgomery exponentiation two calculations are performed upon encountering one bits in the power: a modular multiplication and a modular squaring. These two computations can be represented as:
MMM$(\overline{m}(x), \overline{c}(x)) = \overline{m}(x)\overline{c}(x)r^{-1} \pmod{n(x)}$
$MMM(\overline{m}(x), \overline{m}(x)) = \overline{m}(x)\overline{m}(x)r^{-1} \pmod{n(x)}$
It can be noted that $\overline{m}(x)r^{-1}$ is the common part of both computations. Since there is no data dependency between these steps, the common part can be computed simultaneously and the computations can proceed concurrently.

Returning to our example, let's try both operations (squaring and multiplication by three) concurrently. We can note that $\overline{63} \cdot r^{-1} = 63 \pmod{137}$, therefore: $63 \cdot \overline{63} = 119 \pmod{137}$ $(squaring)$
$63 \cdot \overline{3} = 109 \pmod{137}$ $(multiplication)$ These Montgomery forms are identical to the earlier results.

Bit-level algorithm for parallel Montgomery multiplication and squaring (MMMS):

Input: $\overline{m}(x), \overline{c}(x), n(x)$
Output: $M(x) = MMM(\overline{m}(x), \overline{c}(x))$,
$\quad S(x) = MMM(\overline{m}(x), \overline{m}(x))$
1: $t(x) = \overline{m}(x)$
2: $M(x) = 0, S(x) = 0$
3: for i = k−1 to 0
4: $\quad t(x) = (t(x) + t_0 n(x))/x$
5: $\quad M(x) = M(x) + t(x)\overline{c}_i,$
$\quad S(x) = S(x) + t(x)\overline{m}_i$

And the bit-level algorithm for parallel Montgomery exponentiation:

Input:
$\quad m(x), e, n(x), R_2 = r(x) \cdot r(x) \pmod{n(x)}$
Output: $c(x) = m(x)^e \pmod{n(x)}$
1: $\overline{m}(x) = MMM(m(x), R_2)$
2: $\overline{c}(x) = MMM(1, R_2)$
3: for i = 0 to k−1
4: $\quad$ if e_i == 1:
$\quad (\overline{c}(x), \overline{m}(x)) = MMMS(\overline{m}(x), \overline{c}(x))$
5: $\quad$ else: $\overline{m}(x) = MMM(\overline{m}(x), \overline{m}(x))$
6: $c(x) = MMM(\overline{c}(x), 1)$

The algorithm yields a speed increase on any '1' bits, which should compose about a half of the exponent – the "secret key" – if it is randomly generated.

The original paper [9] focused on parallelizing the exponentiation operation via specialized hardware based on logic gates replicating cellular automata. This paper, however, proposes a software optimization based on an efficient modification of the parallel multiplication and squaring algorithm that can be adapted to popular cryptographic libraries. Testing suggests a significant performance increase that can be leveraged even on common consumer-grade hardware.

VI. Implementation and performance

The implementation was written in Python 3.7 with no external libraries for comparative performance testing of the three algorithms. Python provides long arithmetic natively, making the code much easier to understand for educational purposes. The implementation along with the test results and

further demonstrations is freely available at the following URL: github.com/krikun98/montgomery.

Galois field arithmetic was implemented in accordance with the algorithms shown earlier. Three methods of modular exponentiation were implemented:

1) Left-to-right binary exponentiation [16]
2) Montgomery exponentiation [16]
3) Parallel Montgomery exponentiation [9]

Three sets of performance tests were conducted. The tests were ran on an AMD Threadripper 3960X CPU. The first set of tests utilized small irreducible polynomials – $2 < k < 32$ [17]. This test measured the time per 10000 modular exponentiations. 10000 random numbers and the power were randomly generated for each round.

$$M, E, i = 1 \; to \; 10000 : \; r(x)/x^2 \leq m_i, E < r(x)/x$$

Then, exponents were computed for each number, and the overall time was recorded for each method. Multiple rounds were conducted for each polynomial to ensure the results are statistically valid. The results are depicted on figure 1.

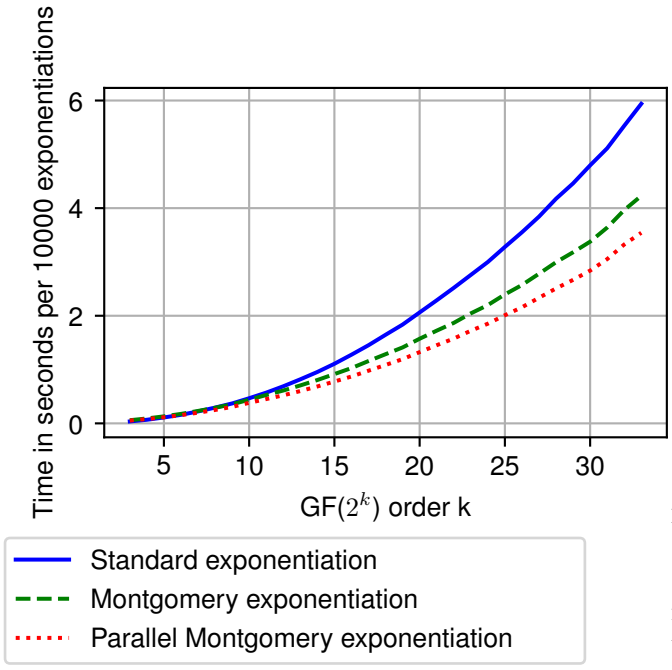


Fig. 1. Performance comparison with small orders.

This set of tests shows even though standard exponentiation is faster for polynomials of order $k \leq 6$, Montgomery exponentiation becomes quicker after that. This is due to the computational overhead incurred by the transformation to the Montgomery form and back from it. The number where Montgomery exponentiation becomes quicker is not precise and may depend on the implementation. While the absolute differences per exponentiation are incredibly small here, further tests demonstrate the proportional difference persists.

The next two sets of tests utilized large irreducible polynomials provided by the IETF that are currently in use. Oakley groups 1 and 2 [4] are still in use as of RFC 7296 (2014), albeit not recommended since they are too short for proper security [14]. They were included to provide more data points. Groups 5 and 14 through 18 were taken from RFC 3526 [5], and groups A.1 through A.5 were taken from RFC (Request for Comments) 7919 [6]. These contain polynomials of higher orders and some are recommended by the Internet Engineering Task Force for current implementations. The second set measured the time per single modular exponentiation. Random numbers were generated for each round.

$$m, E : \; r(x)/x^2 \leq m, E < r(x)/x$$

Exponents were then computed and times were recorded for each method. At least 150 rounds were done for each polynomial to ensure statistical significance of the results. The results are depicted on figure 2.

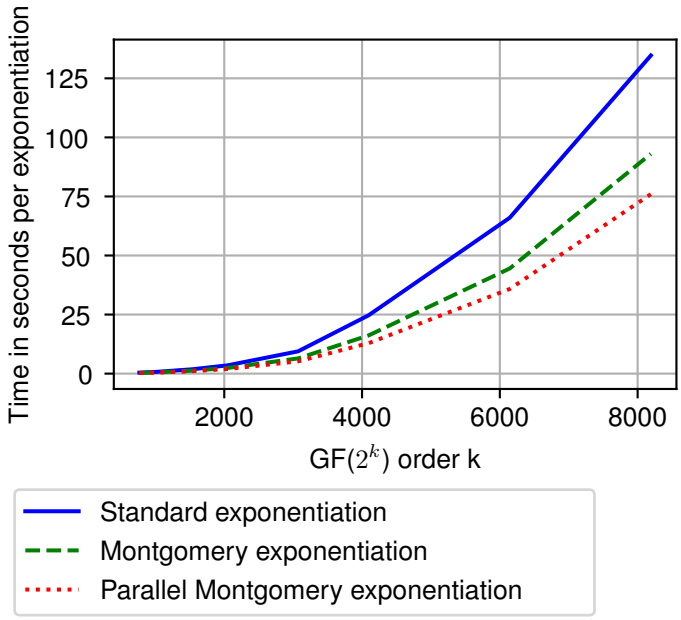


Fig. 2. Performance comparison with RFC polynomials.

The figures demonstrate that the optimized method has nearly the same performance with 4KB orders k as left-to-right exponentiation with 3KB orders. This means the method can offer a significant increase in key length, and, therefore, security with little to no speed penalty. Compared to Montgomery exponentiation, which is used in OpenSSL [3], the gains are less significant, offering a 20 to 25% speed increase.

The third set of tests utilized the same irreducible polynomials as the second, but included all the operations necessary for a Diffie–Hellman key exchange, i.e., picking two random numbers:

$$a, b : \; r(x)/x^2 \leq a, b < r(x)/x$$

And computing $g^a, g^b, (g^a)^b, (g^b)^a$. The generating element $g = 2$ was taken from IETF recommendations. This test offers a further potential speed benefit from keeping the finite field characteristics k, r, r^2 between calculations without needing to re-initialize them.

The absolute times scaled similarly to the previous test case, so the results here are shown in percentages of a

standard (left-to-right) exponentiation. They are depicted on figure 3.

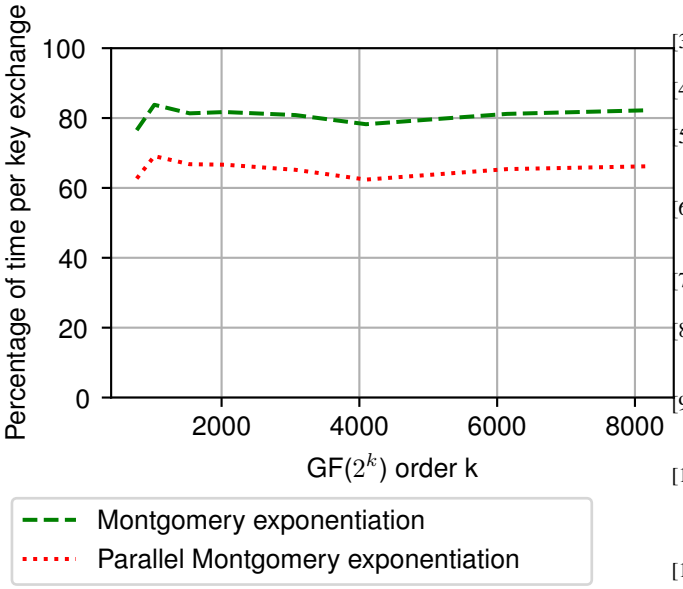


Fig. 3. Performance comparison with DHKE implementations.

These performance figures demonstrate the parallelized Montgomery exponentiation algorithm can yield an up to 40% performance improvement over standard exponentiation and up to a 25% improvement over the unmodified Montgomery algorithm.

VII. Conclusions

This paper presented a description and an evaluation of optimization structure that can yield significant performance improvements in any modular exponentiation workload. An open-source library for Galois field arithmetic containing an implementation of said structure was created for testing and educational purposes.

The optimization structure could potentially be applied not only to Diffie–Hellman, but also to El–Gamal and any other algorithms that rely on modular exponentiation. It could also be applied in existing implementations, yielding a speed increase of up to 25 to 40 percent. The parallelized Montgomery exponentiation algorithm retains all the implementational benefits of Montgomery exponentiation, as well as security-specific advantages such as constant-time execution.

With TLS 1.3 requiring ephemeral keys for the Diffie-Hellman key exchange, as well as utilizing longer groups, the speed of the algorithm comes into focus. The optimization structure discussed here may yield either a significant speed increase or a substantial lengthening of cryptographic keys with little to no impact on performance, depending on the existing implementation. It can provide a stopgap solution before the transition to elliptic curve or post-quantum cryptography.

A direction for further research would be a word-level adaptation of this algorithm, or similar optimizations for elliptic curve cryptography.

References

[1] Bos, Joppe W.; P. Montgomery. Montgomery Arithmetic from a Software Perspective. *IACR Cryptol. ePrint Arch* **2017** (2017): 1057.

[2] Rescorla, E. et al. The Transport Layer Security (TLS) Protocol Version 1.3. Internet Engineering Task Force **2018**, RFC 8446 (Standards Track).

[3] OpenSSL: Cryptography and SSL/TLS toolkit. Available online: https://www.openssl.org/ (accessed on 21.10.20)

[4] Orman, H. The OAKLEY Key Determination Protocol. Internet Engineering Task Force **1998**, RFC 2412 (Informational).

[5] Kivinen, T.; Kojo, M. More Modular Exponential (MODP) Diffie–Hellman groups for Internet Key Exchange (IKE). Internet Engineering Task Force **2003**, RFC 3526 (Standards Track).

[6] Gillmor, D. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). Internet Engineering Task Force **2016**, RFC 7919 (Standards Track).

[7] Knuth, D.E. *The Art of Computer Programming: Seminumerical Algorithms*; vol.2, Addison-Wesley, Reading, MA, Second edition, **1981**.

[8] Koç, Ç.K.; Acar, T. Fast software exponentiation in GF$\left(2^k\right)$. In *Proceedings, 9th Symposium on Computer Arithmetic*, pages 225–231, Asilomar, California, **1997**.

[9] Ku, K.; Ha, K.; Yoo, W.; Yoo, K. Parallel Montgomery multiplication and squaring over GF(2^m) Based on cellular automata. *Proc. ICCSA* **2004**, (LNCS, 3046), pp. 196-205.

[10] Adrian, D., et al. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (CCS '15). Association for Computing Machinery, New York, NY, USA, **2015**, 5–17.

[11] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. In *IEEE Transactions on Information Theory* **1984**, vol. 30, no. 4, pp. 587-594.

[12] CVE-2015-0204, *cve.mitre.org*, . Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0204 (accessed on 25.01.20)

[13] Paar, C.; Adrian, D.; Kasper, E.; Halderman, A.J. et al. DROWN: Breaking TLS using SSLv2. *25th USENIX Security Symposium* **2016**.

[14] Nir, Y.; Kivinen, T. Algorithm Implementation Requirements and Usage Guidance for the Internet Key Exchange Protocol Version 2 (IKEv2). Internet Engineering Task Force **2017**, RFC 8247 (Standards Track).

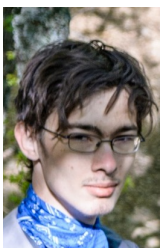
[15] Bluhm, M.; Gueron, S. Fast software implementation of binary elliptic curve cryptography. *J Cryptogr Eng* **2015**, Vol.5, 215–226.

[16] Koç, Ç.K.; Acar, T. Montgomery Multiplication in GF$\left(2^k\right)$. *Kluwer Academic Publishers, Designs, Codes and Cryptography* **1998**, 14(1), pp. 57-69.

[17] Partow, A., "Primitive Polynomial List.", *Partow.net*. Available online: www.partow.net/programming/polynomials/index.html (accessed on 22.10.20).

Alla Levina finished St. Petersburg State University mathematical faculty in 2005, in 2009 got Ph.D in St. Petersburg State University mathematical faculty, Russian Federation.

She is an Associate Professor at Saint Petersburg Electrotechnical University "LETI", Russian Federation. Her research interests include cybersecurity, cryptography, coding theory, and wavelet transformation.



Alexander Krikun finished ITMO University in 2020.

He is a Master's student at ITMO University, Russian Federation. His research interests include cryptography, evolutionary algorithms, and cellular automata.