Generation of Elliptic Curve Points in Tandem

Armando Faz-Hernández[1,2], **Julio López**[1]

[1]Institute of Computing – University of Campinas.
1251 Albert Einstein, Cidade Universitária. Campinas, SP – Brazil.

[2]Cloudflare Inc.
101 Townsend, San Francisco, CA – United States of America.

armfazh@cloudflare.com, jlopez@ic.unicamp.br

Abstract. *A hash to curve function H, mapping bit strings to points on an elliptic curve, is often required in cryptographic schemes based on elliptic curves. Its construction is based on a deterministic encoding and a cryptographic hash function, which complementarily dominate its execution time. To improve the performance of H, we propose a parallel strategy where two units execute in tandem the internal operations of H. We instantiate this approach with a parallel software implementation of a hash to curve function that outputs points on a twisted Edwards curve. A performance benchmark on Haswell and Skylake micro-architectures shows that our parallel implementation is 1.4 times faster than its sequential implementation.*

1. Introduction

A hash function H that maps bit strings to points on an elliptic curve is commonly known as *hashing to curves*. This function plays an important role when instantiating group-based cryptographic algorithms with elliptic curves. A general strategy to define H is to compose a deterministic encoding with a cryptographic hash function.

A deterministic encoding is a rational function that maps field elements to points on an elliptic curve. Several works have proposed deterministic encodings (briefly reviewed in Section 2) that apply to different elliptic curves. Generating points in a deterministic way is preferred over using probabilistic or randomized algorithms, because its implementation is usually easier to protect against side-channel attacks.

Related works on this area focus more on the mathematical construction of deterministic encodings; however, less attention is given to the total cost of H. In a closer inspection, the execution time of H is dominated by the encoding when processing short inputs. This situation holds until an inflection point in the input's length at which the cryptographic hash function becomes the dominant part. Therefore, to reduce the execution time of H both scenarios must be optimized.

In this work, we show how to improve the execution time of hashing to curves through the use of parallel computing. Specifically, we show a general strategy that allows two units to calculate in tandem the internal operations of the hash to curve function H. We instantiate our proposal with a software implementation that uses SIMD vector instructions. We use a twisted Edwards curve introduced by Bernstein and Lange (2007) and derived from Curve25519. Our performance benchmarks show that the vectorized implementation speeds up H when hashing both short and long inputs. The source codes are publicly available at `https://github.com/armfazh/fld-ecc-vec`.

2. Generating Points on Elliptic Curves

Let $\mathbb{F}$ be a finite field of characteristic $p > 3$, and $E/\mathbb{F}\colon y^2 = x^3 + ax + b$ be a Weierstrass curve such that $4a^3 + 27b^2 \neq 0$. The set of rational points of the curve, denoted as $E(\mathbb{F})$, forms an abelian group with identity $\mathcal{O}$. Let q and n denote, respectively, the cardinality of $\mathbb{F}$ and $E(\mathbb{F})$, these numbers are related as $|n - (q + 1)| \leq 2\sqrt{q}$. The group $E(\mathbb{F})$ is finitely generated as the direct sum of cyclic subgroups. In each of these subgroups, there exists a point called generator that has order dividing n.

An element $s \in \mathbb{F}$ is a quadratic residue (QR) in $\mathbb{F}$, if there exists $r \in \mathbb{F}$ such that $s = r^2$, otherwise s is a quadratic non-residue (QNR) in $\mathbb{F}$. Due to Euler's criterion, the function $\chi\colon s \mapsto s^{(p-1)/2}$ gives a method to test whether an element is a quadratic residue on prime fields of odd order p. Thus, given $s \in \mathbb{F} \setminus \{0\}$, it holds $\chi(s) = 1$ if s is QR, and $\chi(s) = -1$ if s is QNR. Let $g(x) = x^3 + ax + b$ be the right-hand side of the curve equation. Clearly, for any $x \in \mathbb{F}$ such that $g(x)$ is QR, it follows that $(x, \pm\sqrt{g(x)}) \in E(\mathbb{F})$.

2.1. Deterministic Encodings

Finding a systematic method of generating non-trivial points on an elliptic curve is a problem solved in many different ways. We summarize some well-known algorithms for generating points.

The simplest method is by trial and error. Given $E(\mathbb{F})$ as an input, first select at random $x \in \mathbb{F}$, which is a candidate to be the x-coordinate of a point, and calculate $g(x)$. If $g(x)$ is QR, calculate $y = \sqrt{g(x)}$, and a point (x, y) has been found. Otherwise, repeat the process of sampling another candidate. The execution time of this randomized algorithm varies at each invocation since it runs until it finds a point on the curve.

The try-and-increment method shown by Boneh and Franklin (2001) does not require random number generation. Instead, this method takes as an input $u \in \mathbb{F}$, and an integer k. First, start a counter $i = 0$ and declare $x = u + i$ as the candidate value for the x-coordinate. Similarly to the previous method, determine whether $g(x)$ is QR; if so, calculate $y = \sqrt{g(x)}$, and a point (x, y) has been found. Otherwise, increment the counter and repeat the procedure until $i = k$. Unlike the previous method, this algorithm can fail after k iterations with probability 2^{-k}. Although its execution time still varies, the time is bounded because the algorithm performs at most k iterations. Thus, increasing k reduces the chances of failure, but also increases its execution time.

A better approach is to use a non-randomized algorithm such as the one shown by Skałba (2005). This method defines rational functions $X_i(u)$ defined over $\mathbb{F}$ such that $X_4(u)^2 = g(X_1(u)) \cdot g(X_2(u)) \cdot g(X_3(u))$. For those $u \in \mathbb{F}$ that satisfy this equation, it is guaranteed that exists at least one index $i \leq 3$ such that $g(X_i(u))$ is QR. Consequently, there are at most three candidates for the x-coordinate of a point. This method is deterministic since always finds a point (x, y) on the curve given as $x = X_i(u)$ and $y = \sqrt{g(X_i(u))}$. A downside of Skalba's functions is their large degree. Later, Shallue and van de Woestijne (2006) simplified these functions providing an efficient algorithm for constructing points on an elliptic curve.

Several works followed a similar approach for constructing points. The general recipe of these constructions, called *deterministic encodings*, is to formulate rational maps that produce valid candidates for the coordinates of a point. The number of candidates

is always a small, fixed number. In the best case, there is only one candidate, such as in the algorithm by Icart (2009). None of these methods requires randomization, and they always produce a point given $u \in U \subseteq \mathbb{F}$, where U is the domain of the rational functions.

In the literature, there exist several deterministic encodings with optimizations tailored for specific elliptic curve models. For instance, the Simplified SWU encoding, proposed by Brier et al. (2010), applies to elliptic curves defined over fields of characteristic $p \equiv 3 \pmod 4$. Bernstein et al. (2013) introduced the Elligator 2 encoding that applies to Montgomery curves. This encoding extends to twisted Edwards curves using the birational map existent between these curves. Fouque and Tibouchi (2012) showed an encoding for the Barreto-Naehrig pairing-friendly curves. Work in progress at the Internet Engineering Task Force (IETF) by Faz-Hernández et al. (2020) focuses on a standard specification of hash to curve methods.

Another approach relies on having previous knowledge of the generator points of the curve. Let $E(\mathbb{F}) = \langle P_1 \rangle \times \cdots \times \langle P_t \rangle$ for some $t > 0$. Hence, any point $P \in E(\mathbb{F})$ can be written as a linear combination of the generator points $P = \sum_{i=1}^{t} k_i P_i$. Hence, one can obtain a point on the curve sampling scalars k_i at random in the range $0 \leq k_i < \text{ord}(P_i)$. Although this method is also deterministic, it is not as efficient as any of the previous methods, because scalar multiplications are more expensive than square roots.

2.2. Indifferentiable Hashing to Elliptic Curves

Let $H\colon \{0,1\}^* \to E(\mathbb{F})$ be a function that maps bit strings to points on an elliptic curve. A common construction of H is as a function composition of a deterministic encoding $f\colon \mathbb{F} \to E(\mathbb{F})$ with a hash function $h\colon \{0,1\}^* \to \mathbb{F}$. Unfortunately, $f \circ h$ cannot be used as a secure hash function since its output is not uniform.

Maurer et al. (2004) showed that those cryptographic schemes proved secure in the random oracle model require that hash functions must be indifferentiable from a random oracle. Brier et al. (2010) proved that

$$H(m) = f(h_0(m)) + f(h_1(m)) \tag{1}$$

is indifferentiable whenever f is the Icart's encoding, and h_0 and h_1 are independent hash functions into $\mathbb{F}$. Later, Farashahi et al. (2013) generalized this construction for any deterministic encoding f whenever f is a well-distributed encoding. Relying on these results, Eq. (1) provides a secure way to instantiate a hash to curve function H that is indifferentiable from a random oracle.

3. Parallel Hashing

We are interested in accelerating the execution time of H given in Eq. (1). Note that the input's length can grow arbitrarily, so the evaluation of h_0 and h_1 can dominate the execution time of H. However, if its length is short, the execution time of f is what dominates the computation.

Given an input string m, we can decompose the calculation of $H(m)$ in two parts. From Eq. (1), it is clear that $P_0 = f(h_0(m))$ and $P_1 = f(h_1(m))$ are independent computations, and thus, they can be performed in parallel. This workload splits evenly into two sub-tasks, whose output points are then sequentially added to produce the final result.

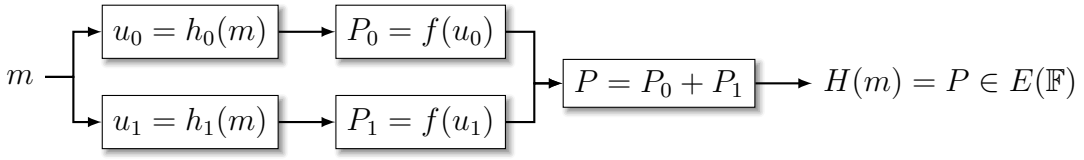

Figure 1. Parallel hashing a bit string m into a point P on an elliptic curve.

As it is shown in Figure 1, the calculation of H exhibits some degree of data-level parallelism that allows scheduling operations independently. An advantage of this construction is that it applies in general regardless of the hash function or encoding used. Moreover, the parallel calculation benefits both processing short and long inputs.

To demonstrate this approach, we developed a software implementation using SIMD vector units. In the following sections, we briefly introduce vector units, describe the elliptic curve instance chosen, and give details of the implementation.

3.1. Parallel SIMD Computing

Hardware optimizations targeting data-level parallelism are usually present in the form of extensions to the instruction set architecture (ISA). Thus, programmers explicitly instrument code using special instructions assigning data to the parallel units for its processing.

Vector units are extensions to the ISA that execute an instruction over sets of data simultaneously following the SIMD parallel computing paradigm. The common usage of these extensions is to load data into vector registers (a set of words) and use vector instructions for performing an operation over each word stored in the vector registers. For example, let `X0` and `X1` be vector registers of 256-bits (storing eight 32-bit words), the vector instruction `VADD X0 X1` will perform eight 32-bit additions of the words stored in `X0` and `X1`.

An advantage of vector instructions is that they reduce the size of programs by encoding in one instruction several repetitive instructions. Thus, shorter instruction encodings help to release pressure on instruction decoders as decoding occurs once for many operations. In most architectures, vector instructions usually run as fast as sequential instructions however they process more data. However, they also fetch more data from memory, which tends to be a costly operation.

Vector units are available in several computer architectures. For instance, the AMD64 (also known as x64-86) architecture includes a vast number of vector instructions named SSE and AVX (Intel Corporation, 2011). The ARM architecture also contains an extension called ASIMD vector unit formerly known as the NEON vector unit (ARM, 2009). Similarly, the Power ISA architecture supports the AltiVec (Freescale Semiconductor, 1999) vector instructions. Although these extensions are not compatible with each other, they contain equivalent instructions that perform arithmetic and logic operations. Their presence on several computing devices, such as conventional processors and Internet servers, motivates their application on several computing fields.

3.2. Elliptic Curve Instance

Among the options for our implementation, we choose `edwards25519` that is a secure and well-known elliptic curve instance. It is a twisted Edwards curve derived from

Curve25519 by Bernstein and Lange (2007) and is defined over $\mathbb{F}_p$, where $p = 2^{255} - 19$, by the equation

$$\texttt{edwards25519:}\ -x^2 + y^2 = 1 - \tfrac{121665}{121666} x^2 y^2 \,. \tag{2}$$

The set of rational points $E(\mathbb{F}_p)$ is a group of order $n = hr$, where the cofactor is $h = 8$ and $r = 2^{252} + 27742317777372353535851937790883648493$ is prime.

A suitable deterministic encoding for twisted Edwards curves is Elligator 2 introduced by Bernstein et al. (2013). Although Elligator 2 applies to Montgomery curves, the point produced by Elligator 2 can be mapped to the twisted Edwards curve using the birational map Φ that connects these two curves. Hence, the deterministic encoding $f\colon \mathbb{F}_p \to E(\mathbb{F}_p)$ is defined as the function composition of Elligator 2 with Φ. Algorithm 1 shows this deterministic encoding explicitly.

Algorithm 1 Deterministic encoding f for twisted Edwards curves.

Input: $u \in \mathbb{F}_p$ and β is a fixed QNR in $\mathbb{F}_p$.
Output: $f(u) = (x, y) \in E(\mathbb{F}_p)$.

{Elligator 2 encoding to the Montgomery curve: ${y_0}^2 = {x_0}^3 + A{x_0}^2 + x_0$}
1: $v = -A/(1 + \beta u^2)$
2: $e = \chi(v^3 + Av^2 + v)$
3: $x_0 = ev - (1 - e)A/2$
4: $y_0 = -e\sqrt{{x_0}^3 + A{x_0}^2 + x_0}$

{Φ mapping to the twisted Edwards curve: $ax^2 + y^2 = 1 + dx^2y^2$}
5: $x = x_0/y_0$
6: $y = (x_0 - 1)/(x_0 + 1)$
7: **return** (x, y)

*For `edwards25519`, use $p = 2^{255} - 19$, $A = 486662$, $\beta = 2$, and return $(\sqrt{-486664}\, x, y)$.

The last part of H adds the points produced by the two invocations of the encoding. Recall that some twisted Edwards curves have a $\mathbb{F}_p$-complete formula for calculating point additions. The completeness of the formula is particularly useful since the order of the points produced by f is arbitrary. Thus, the addition of twisted Edward points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ is calculated as

$$P + Q = \left(\frac{x_P y_Q + y_P x_Q}{1 + d x_P y_P x_Q y_Q}, \frac{y_P y_Q - a x_P x_Q}{1 - d x_P y_P x_Q y_Q} \right) . \tag{3}$$

In the implementation, we calculate the point addition and the encoding using the extended projective coordinate system introduced by Hisil et al. (2008). Thus, all point operations are performed without field inverses, except by the one used to convert the output point to affine coordinates. Note that in some situations this conversion might not be needed if the output point is part of a further operation such as a scalar multiplication.

The hash functions h_0 and h_1 internally use SHA-512 by NIST (2008). A common strategy for deriving two different oracles from a single hash function is to concatenate a domain separation string. A detailed specification for securely instantiating these functions is given in the IETF draft (2020).

3.3. Implementation Details

To determine the performance savings introduced by the parallel strategy proposed, we developed both a sequential and a parallel implementation of the function H.

For the sequential implementation, we build it on top of the optimized library by Oliveira et al. (2018) for calculating the prime field operations in Algorithm 1. For SHA-512, we use a 64-bit implementation by Pornin, which is available at eBACS (2020).

The parallel implementation of SHA-512 is described as follows. Most of the operations of SHA-512 are over 64-bit words, and consequently, using 128-bit vector registers allows carrying the calculation of two SHA-512 evaluations simultaneously. This workload is easy to implement with vector instructions because it always operates on fixed-size operands.

On the other hand, implementing prime field operations with vector instructions is more elaborated. Operating elements of $\mathbb{F}_p$ requires handling large integers using, for example, a multi-precision library such as GMP (2012). It is common that multi-precision libraries process only one field operation at a time rather than several field operations in parallel. In the face of this lack of support, the vectorized library by Faz-Hernández et al. (2019) comes in useful since it supports the parallel execution of field operations through AVX2 vector instructions. We use this library to implement Algorithm 1 scheduling two field operations in parallel.

Some parts of H cannot run in parallel. For example, adding[1] the points produced by each encoding invocation, or converting points from projective to affine coordinates. These tasks limit the theoretical speedup factor of the parallel implementation.

3.4. Performance Evaluation

We measured the latency of both implementations of the function H that outputs points on the `edwards25519` curve. Table 1 lists the execution times of the sequential and parallel implementations running on Haswell and Skylake micro-architectures, together with a breakdown of the internal operations of H.

Table 1. Latency (in 10^3 clock cycles) of hashing 64-byte strings to points on the `edwards25519` curve.

Description	Operation	Haswell			Skylake		
		Seq.	Par.	Δ	Seq.	Par.	Δ
Hash	$u_i = h_i(m)\,\vert_{i=0,1}$	3.3	1.9	43.2 %	3.3	1.8	44.9 %
Encoding	$P_i = \Phi(f(u_i))\,\vert_{i=0,1}$	35.2	19.2	45.4 %	27.6	15.7	43.0 %
Point addition	$P = P_0 + P_1$	0.6	0.6	–	0.5	0.5	–
Point to affine	$(x, y) = P$	15.1	15.1	–	11.9	11.9	–
Hash to curve	$H(m)$	57.5	40.4	29.7 %	46.0	32.7	28.8 %

The Δ column represents the percentage of improvement of hashing with the parallel implementation versus the sequential implementation. Haswell is a Core i7-4770 processor, and Skylake is a Core i7-6700K processor. Measurements were taken disabling the frequency scaling and multi-threading modes, and the source codes were compiled with Clang v5.0.1.

[1]Note that it is possible to run the internal operations of the point addition formula in parallel.

The execution times shown in Table 1 correspond to hashing short inputs. As can be seen, the cost of SHA-512 only represents a small fraction of the total cost of H; on the other hand, the cost of the deterministic encoding amounts 60 % of the total. Hence, it is expected that executing the encoding in parallel reduces the execution time of H to a great extent. According to our measurements, the vectorized implementation is 29 % faster than the sequential implementation, and this improvement is consistent in both architectures.

Another metric to compare our implementation with related works is the time consumed by the Elligator 2 map. Aranha et al. (2014) reported 36,590 cycles on Haswell to calculate the map. Using the Sodium library by Denis (2019), we measured 88,500 cycles on Haswell. In the same architecture, our sequential implementation takes 35,200 cycles calculating two map evaluations. Moreover, our vectorized implementation outperforms all these timings since it takes 19,200 cycles executing two Elligator 2 maps in parallel. This speedup is mainly due to the time saved by performing field exponentiations in parallel. These exponentiations are required to calculate inversions, square-roots, and testing quadratic residuosity.

We also measured the execution time of H as a function of the input's length. Figure 2 shows the speedup factor of the parallel implementation over the sequential implementation. From the graph, one can see that running the deterministic encoding in parallel yields a speedup factor of 1.40. This factor holds for inputs shorter than 1 KB, which corresponds to the block size of SHA-512. However, when hashing longer inputs most of the time is spent on calculating SHA-512 in parallel, and the speedup factor increases getting closer to the ideal factor (2.0).

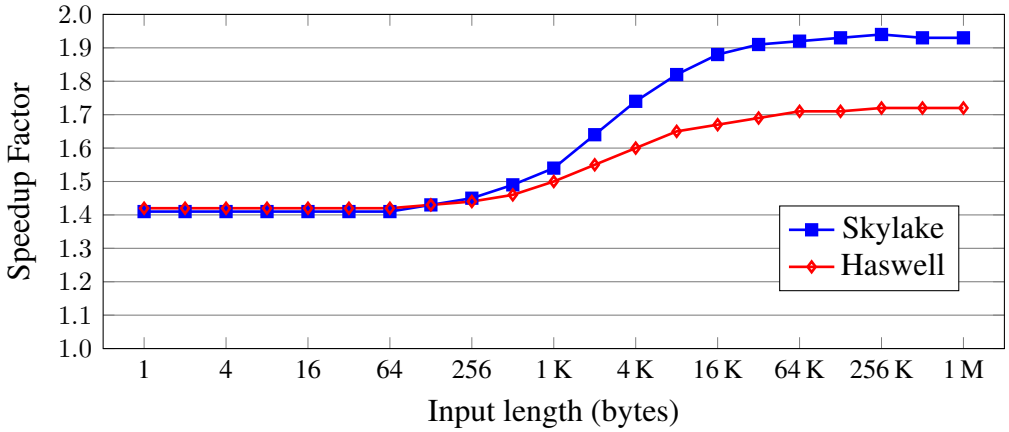

Figure 2. Speedup factor for hashing to `edwards25519` obtained by the vectorized implementation over the sequential implementation.

As can be seen, there is a gap on the speedup factors yielded by Haswell and Skylake. When SHA-512 dominates the execution (long inputs), the processor executes mostly binary operations, as opposed to execute integer additions and multiplications required by the prime field operations when the encoding dominates (short inputs). Skylake has more vector execution units than Haswell, which explains the gap on speedup when processing SHA-512. A similar gap also exists in the execution of prime field arithmetic; however, it was counteracted because the sequential implementation is also accelerated by the ADX instructions available in Skylake.

4. Final Remarks

A hash to curve function H outputting points on an elliptic curve presents two scenarios. When the input is shorter than a certain threshold, the deterministic encoding dominates its execution time; otherwise, the cryptographic hashing function is the dominant part.

We presented a parallel strategy for improving the execution time of H. To do that, we leveraged the data-level parallelism in its definition in such a way that its time-consuming operations run in parallel. We experimentally verified our approach with a vectorized software implementation. In a performance benchmark, the vectorized implementation is $1.40\times$ faster than its sequential counterpart when processing short-sized inputs. Moreover, this speedup factor increases when hashing inputs longer than the block size of the underlying hash function. These results provide evidence that the parallel strategy proposed improves the performance of H for short and long inputs.

A path for further investigation is the application of the proposed methodology to different computing environments, such as hardware implementations or vector units available in other architectures. It also remains as future work to investigate the improvements on hashing when other elliptic curves or deterministic encodings are used.

References

Aranha, D. F., Fouque, P.-A., Qian, C., Tibouchi, M., Zapalowicz, J.-C. (2014). Binary Elligator Squared. In A. Joux A. Youssef (Eds.), *Selected Areas in Cryptography – SAC 2014* (Vol. 8781, pp. 20–37). Springer, Cham. doi: 10.1007/978-3-319-13051-4_2

ARM Limited. (2009, June). Introducing NEON TM development article (Computer software manual No. 1). Retrieved from `http://infocenter.arm.com/help/topic/com.arm.doc.dht0002a`

Bernstein, D. J., Hamburg, M., Krasnova, A., Lange, T. (2013). Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC conference on computer & communications security* (pp. 967–980). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2508859.2516734

Bernstein, D. J., Lange, T. (2007). Faster addition and doubling on elliptic curves. In K. Kurosawa (Ed.), *Advances in Cryptology – ASIACRYPT 2007* (Vol. 4833, pp. 29–50). Springer Berlin Heidelberg. doi: 10.1007/978-3-540-76900-2_3

Bernstein, D. J., Lange, T. (2020, March). *ebacs: Ecrypt benchmarking of cryptographic systems.* Accessed on March 2020. Retrieved from `http://bench.cr.yp.to/supercop.html`

Boneh, D., Franklin, M. (2001). Identity-based encryption from the Weil pairing. In J. Kilian (Ed.), *Advances in Cryptology — CRYPTO 2001* (Vol. 2139, pp. 213–229). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/3-540-44647-8_13

Brier, E., Coron, J.-S., Icart, T., Madore, D., Randriam, H., Tibouchi, M. (2010). Efficient indifferentiable hashing into ordinary elliptic curves. In T. Rabin (Ed.), *Advances in Cryptology – CRYPTO 2010* (Vol. 6223, pp. 237–254). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-14623-7_13

Denis, F. (2019, May). Sodium (v1.0.18 ed.) [Computer software]. Retrieved from `http://libsodium.org`

Farashahi, R. R., Fouque, P.-A., Shparlinski, I. E., Tibouchi, M., Voloch, J. F. (2013). Indifferentiable deterministic hashing to elliptic and hyperelliptic curves. *Mathemathics of Computation*, *82*(281), 491–512. doi: 10.1090/S0025-5718-2012-02606-8

Faz-Hernández, A., López, J., Dahab, R. (2019, July). High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Transactions on Mathematical Software (TOMS)*, *45*(3), 1–35. doi: 10.1145/3309759

Faz-Hernández, A., Scott, S., Sullivan, N., Wahby, R. S., Wood, C. A. (2020, June 29). *Hashing to elliptic curves* (Internet-Draft No. draft-irtf-cfrg-hash-to-curve-09). Internet Engineering Task Force. Retrieved from `https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve` (Work in Progress)

Fouque, P.-A., Tibouchi, M. (2012). Indifferentiable hashing to Barreto–Naehrig curves. In A. Hevia G. Neven (Eds.), *Progress in Cryptology – LATINCRYPT 2012* (Vol. 7533, pp. 1–17). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-33481-8_1

Freescale Semiconductor. (1999, June). AltiVec Technology Programming Interface Manual [Computer software manual]. Retrieved from `https://www.nxp.com/docs/en/reference-manual/ALTIVECPIM.pdf`

Granlund, T., the GMP development team. (2012). GNU MP: The GNU Multiple Precision Arithmetic Library (5.0.5 ed.) [Computer software]. Retrieved from `http://gmplib.org/`

Hisil, H., Wong, K.-H., Carter, G., Dawson, E. (2008). Twisted Edwards curves revisited. In J. Pieprzyk (Ed.), *Advances in Cryptology - ASIACRYPT 2008* (Vol. 5350, pp. 326–343). Springer Berlin Heidelberg. doi: 10.1007/978-3-540-89255-7_20

Icart, T. (2009). How to hash into elliptic curves. In S. Halevi (Ed.), *Advances in Cryptology - CRYPTO 2009* (Vol. 5677, pp. 303–316). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-03356-8_18

Intel Corporation. (2011, June). Intel Advanced Vector Extensions Programming Reference (Vol. 319433-011) [Computer software manual]. Retrieved from `https://software.intel.com/sites/default/files/m/f/7/c/36945`

Maurer, U., Renner, R., Holenstein, C. (2004). Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In M. Naor (Ed.), *Theory of cryptography* (Vol. 2951, pp. 21–39). Springer Berlin Heidelberg. doi: 10.1007/978-3-540-24638-1_2

National Institute of Standards and Technology. (2008). *FIPS PUB 180-4, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-4*. Gaithersburg, MD, USA: National Institute for Standards and Technology. (Supersedes FIPS PUB 180-3 October 2008.) doi: 10.6028/NIST.FIPS.180-4

Oliveira, T., López, J., Hışıl, H., Faz-Hernández, A., Rodríguez-Henríquez, F. (2018). How to (pre-)compute a ladder. In C. Adams J. Camenisch (Eds.), *Selected Areas in Cryptography – SAC 2017* (Vol. 10719, pp. 172–191). Springer, Cham. doi: 10.1007/978-3-319-72565-9_9

Shallue, A., van de Woestijne, C. E. (2006). Construction of rational points on elliptic curves over finite fields. In F. Hess, S. Pauli, M. Pohst (Eds.), *Algorithmic number theory* (Vol. 4076, pp. 510–524). Springer Berlin Heidelberg. doi: 10.1007/11792086_36

Skałba, M. (2005). Points on elliptic curves over finite fields. *Acta Arithmetica*, *117*(3), 293–301. Retrieved from `http://eudml.org/doc/278782`