Optimized Discrete Logarithm Computation for Faster Square Roots in Finite Fields

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

4 June, 2023

Abstract. For computing square roots in a finite field $GF(q)$ where $q - 1 = 2^n m$ for an odd integer m and some integer n, the classic Tonelli-Shanks algorithm starts with an exponentiation (the exponent has size about $\log_2 q - n$ bits), followed by a discrete logarithm computation in the subgroup of 2^n-th roots of unity in $GF(q)$; the latter operation has cost $O(n^2)$ multiplications in the field, which is prohibitive when n is large. Bernstein proposed an optimized variant with lookup tables, leading to a runtime cost of $O((n/w)^2)$, using w-bit tables of cumulative size $O(2^w n/w)$. Sarkar recently improved on the runtime cost, down to $O((n/w)^{1.5})$, with the same overall storage cost. In this short note, we explore the use of a straightforward divide-and-conquer variant of the Pohlig-Hellman algorithm, bringing the asymptotic cost down to $O(n \log n)$, and further study some additional optimizations. The result appears to be competitive, at least in terms of number of multiplications, for some well-known fields such that the 224-bit field used in NIST standard elliptic curve P-224 (for which $n = 96$).

1 Introduction

Let $GF(q)$ be a finite field such that $q = 1 + 2^n m$, for an odd integer m, and integer $n \geq 1$ (q is not necessarily prime; everything in this note also applies to field extensions). Extracting square roots in $GF(q)$ is easy when $n = 1$, in which case $q = 3 \bmod 4$ and it suffices to raise the input x to the power $(q + 1)/4$. When $n = 2$, the field order is $q = 5 \bmod 8$ and Atkin's formulas[2] can be used. For larger n, one may apply the classic algorithm described first by Tonelli[12] then rediscovered by Shanks[9]; it uses the following steps:

1. If input x is zero, then return zero.
2. Let $v = x^{(m-1)/2}$, then $w = xv$ and $u = wv$. This implies that $u = x^m$, and thus $u^{2^n} = x^{q-1} = 1$.
3. Let g be a fixed, known generator of the cyclic group of the 2^n-th root of 1 in $GF(q)$. Find integer e (with $0 \leq e < 2^n$) such that $u = g^e$.
4. If e is odd, then x is not a quadratic residue. Otherwise, a square root of x is $w \cdot g^{(2^n - e)/2}$.

Step 3 is really a specialized version of the discrete logarithm problem, in base g. This problem is computationally "easy" since the order of g is 2^n, which is a very smooth integer; however, it is still an expensive step, especially if n is large. In all of the following, we consider only the number of multiplications in $GF(q)$ as the measure of runtime cost, with some provision for squarings, whose individual cost is typically about 80% of that of generic

multiplications[1]. We will use the notation "xS + yM" to designated a cost of x squarings and y generic multiplications.

Tonelli's algorithm, for step 3, uses a bit-by-bit process, which can be described recursively:

- If $u^{2^{n-1}} = 1$, then e is even, otherwise e is odd. Set $e_0 = e \bmod 2$ accordingly.
- Set $u' = u/g^{e_0}$ and solve (recursively) $u' = (g^2)^{e_1}$ for exponent e_1 (since u' has then order 2^{n-1}, and g^2 is a generator of the 2^{n-1}-th roots of 1).
- Return $e_0 + 2e_1$.

This process can be organized in an iterative way (really, tail-recursive). Division by g^{e_0} can be replaced with multiplication by g^{1-e_0}, provided that the returned exponent is then $e_0 + 2e_1 - 1 \bmod 2^n$. The generator g is supposed to have been chosen in advance; such a generator can be found by choosing a non-square element in $GF(q)$, and raising it to the power m (which generator is selected does not matter for the rest of this paper). Similarly, successive squarings of g (g^2, g^4...) can be precomputed. That process requires about $n^2/2$ multiplications (most of which being squarings). If the input x is not secret, and thus the implementation does not need to be constant-time, then (on average) half of the recursive invocations can be skipped, by noticing that if $u^{2^{n-j}} = 1$, then e is a multiple of 2^j and we can use $u' = (g^{2^j})^{e_j}$; the cost is then lowered to about $n^2/4$ multiplications.

This cost, being quadratic in n, can become prohibitive when n is large. A well-known case is when working in the finite field over which the standard NIST elliptic curve P-224 is defined[6]; that field uses $q = 2^{226} - 2^{96} + 1$, i.e. $n = 96$, in which case the cost of the Tonelli-Shanks algorithm is about 2500 field multiplications. A square root is used when decoding a public key (curve point) from its compressed format, and that single square root operation is then close, in runtime cost, to that of a generic curve point multiplication by a scalar; thus, the overhead on, for instance, signature verification, is non-negligible. Similarly inconvenient fields are used in curve cycles for recursive proof composition systems, where a large n value improves polynomial arithmetics[4].

In the Tonelli-Shanks algorithm, the input value (u) is modified bit-by-bit (in low-to-high order) so as to become a 2^j-th root of 1 for decreasing values of j. The Adleman-Manders-Miller algorithm[1] is a variant in which the modifications of the input u are instead accumulated separately; this variant has complexity quadratic in n, and is on average somewhat slower than Tonelli-Shanks.

Bernstein improved on the Tonelli-Shanks performance with the use of precomputed tables[3]. The method is an extension of the discrete logarithm part, to process w bits at a time, for some parameter w. Namely, $u^{2^{n-w}}$ is a 2^w-th root of 1, and these 2^w roots can be precomputed; the discrete logarithm of such a root can then be obtained with a reverse-lookup in a table of these roots. Moreover, precomputed tables of powers of g (n/w tables of 2^w elements, with table i containing values g^{iw+j} for $j = 0$ to $2^w - 1$) speed up the other operations. Bernstein's method achieves cost $O((n/w)^2)$ field multiplications, with a storage cost

[1] That ratio can vary depending on the field and the implementation, but since a multiplication can be computed with two squarings[13] with $ab = ((a+b)/2)^2 - ((a-b)/2)^2$, squaring cost cannot really be less than half of that of a multiplication.

of $O(2^w n/w)$ field elements for the tables[2]. The P-224 field is explicitly used as example by Bernstein, with $w = 6$: precomputed tables then contain 64 field elements each (1024 in total, about 29 kB), and the total cost is 90S + 138M (not counting the cost of the initial exponentation with power $(m-1)/2$, which can be done in cost 126S + 10M for that field).

More recently, Sarkar proposed an improved variant of Bernstein's method[8]. While Bernstein's algorithm can be viewed as a w-bit extension of Tonelli-Shanks, Sarkar's proposal is a combination of such a variant, and of a w-bit extension of Adleman-Manders-Miller. Sarkar showed that with a careful choice of how to do that combination, a sub-quadratic complexity can be achieved, namely $O((n/w)^{1.5})$ field multiplications, for the same table storage cost as in Bernstein's algorithm ($O(2^w n/w)$ field elements). While the subquadratic complexity necessarily implies a lower cost for large enough values of n, it is not necessarily better for a specific, fixed n. With the same example used by Bernstein (P-224's field with $n = 96$, and using $w = 6$), Sarkar's method has a 146S + 82M cost (not counting the initial exponentiation), which is the exact same number of field operations as with Bernstein's method, albeit with a higher squaring-to-multiplications ratio, thus offering slightly better performance if the implementation of squarings happens to be faster than that of general multiplications.

In this note, we explore a different variant. The Tonelli-Shanks algorithm can be viewed as a sub-case of the Pohlig-Hellman algorithm for solving discrete logarithms with smooth order[7]. The Pohlig-Hellman algorithm handles order powers p^a one digit (in base p) at a time, like Tonelli-Shanks, which can be viewed as a split of p^a into p and p^{a-1}; this admits a natural extension into a more balanced split into two $p^{a/2}$ halves. This "divide-and-conquer" strategy leads to a $O(n \log n)$ cost[3]. Applied to the specific problem of computing square roots in finite fields, the same $O(n \log n)$ complexity is obtained, which is asymptotically better than both Bernstein's and Sarkar's methods. Precomputed tables can moreover help speed up some parts of the algorithm. With the same example as previously (P-224's field, with $n = 96$), and using 6-bit tables ($w = 6$, for the same overall storage cost in $O(2^w n/w)$ field elements), we reduce the cost to 142S + 60M (still not counting the initial exponentation), which has a lower total number of operations (202 instead of 228), and an even higher squaring-to-multiplications ratio. In practical terms, the performance difference is minor, but it becomes significant for larger values of n.

The following sections describe our algorithm and discuss a few extra optimizations. An heavily commented example implementation (using Sage) is available at:

https://github.com/pornin/modsqrt

2 The Improved Method

Algorithm 1 is the direct transcription of the generic method described in the previous section, which is common to Tonnelli-Shanks, Adleman-Manders-Miller, Bernstein, Sarkar and the new method. It uses DLPpow2 (algorithm 2), which is the Pohlig-Hellman variant improved with the divide-and-conquer strategy. We stress out that neither algorithm is novel;

[2] All precomputed tables are prepared in advance, and are not modified when used, so that, in an embedded system context, they can be stored as ROM/Flash, which is a less scarce resource than RAM.

[3] This was already noted by Shoup ([10], section 11.2.3), but Shoup does not claim authorship of that idea, which may have been expressed previously.

only the combination of the two for the purpose of extracting square roots (which was apparently previously overlooked) and the various optimization techniques discussed thereafter.

Algorithm 1 Square Root Extraction in $GF(q)$

Input: $x \in GF(q)$, with $q = 2^n m + 1$ ($m = 1 \bmod 2$, $n \geq 1$)
Output: $y \in GF(q)$ such that $y^2 = x$, or $\perp$ if x is not a square in $GF(q)$

if $x = 0$ **then**
    return 0
$v \leftarrow x^{(m-1)/2}$
$w \leftarrow xv$
$u \leftarrow wv$
$e \leftarrow \mathsf{DLPpow2}(u, g, n)$
if $e = 0 \bmod 2$ **then**
    return $w \cdot g^{(2^n - e)/2}$
else
    return $\perp$

Algorithm 2 DLPpow2: Discrete Logarithm among 2^n-th Roots of 1 in $GF(q)$

Input: $u, g \in GF(q)$ and $n \geq 1$ such that $g^{2^{n-1}} = -1$ and $u^{2^n} = 1$
Output: e such that $0 \leq e < 2^n$ and $u = g^e$

if $n = 1$ **then**
    if $u = 1$ **then**
        return 0
    else
        return 1
$a \leftarrow \lfloor n/2 \rfloor$
$b \leftarrow n - a$
$c \leftarrow \mathsf{DLPpow2}(u^{2^b}, g^{2^b}, a)$
$d \leftarrow \mathsf{DLPpow2}(u \cdot g^{2^a - c}, g^{2^a}, b)$
return $c + (d - 1 \bmod 2^b)$

Soundness. The reason the algorithm always works is that 2^n-th roots of 1 in a finite field $GF(q)$ are a cyclic group. Thus, if g is a generator of that group, then g^{2^j} is a generator of the 2^{n-j}-th roots of 1 and has order exactly 2^{n-j}. If $u = g^e$, then $u^{2^b} = (g^e)^{2^b} = (g^{2^b})^e$, therefore the first recursive call must yield e modulo the order of g^{2^b}, i.e. $c = e \bmod 2^a$. If we write $e = c + 2^a f$, then $0 \leq f < 2^b$, and:

$$\begin{aligned} u \cdot g^{2^a - c} &= g^c g^{2^a f} g^{2^a} g^{-c} \\ &= (g^{2^a})^{f+1} \end{aligned}$$

which means that the second recursive call returns $d = f + 1$ modulo the order of g^{2^a}, which is 2^b. We thus get $f = d - 1 \bmod 2^b$, which yields the complete value of f given its possible range

of values. The use of a *multiplication* of u by g^{2^a-c}, instead of a *division* of u by g^c, promotes efficiency because multiplications are typically substantially faster than divisions in finite field implementations, but requires a corrective action, which is here the (inexpensive) subtraction of 1 from the obtained d value. In some finite fields, we can use a plain division by g^c instead: if $q = p^2$ for a prime $p = 3 \bmod 4$, then $GF(q)$ is a degree 2 extension of $GF(p)$ that can be defined with a symbolic element $i = \sqrt{-1}$. In that case, the inverse of a 2^n-th root of 1 in $GF(q)$ is simply its conjugate, i.e. is obtained by negating the "imaginary part"; thus, for such a field, the cost of dividing by g^c is about the same as the cost of a single multiplication in the field.

Complexity. Assuming no precomputed values, the computations of g^{2^b}, u^{2^b}, and g^{2^a} are done in at most $2b \leq n + 1$ squarings in total, while $u \cdot g^{2^a-c}$ needs at most $2a - 1 \leq n - 1$ multiplications (with a square-and-multiply algorithm, noting that $2^a - c$ fits on a bits, unless $c = 0$, in which case $g^{2^a-c} = g^{2^a}$, which we also compute and can thus reuse in that case). This is a total of at most $2n$ multiplications. The two recursive calls are for parameters a and b, whose sum is n; thus, the total cost must be lower than $2nh$, with h being the recursion depth, which is necessarily at most $\lceil \log_2 n \rceil$ (recursion depth is easily seen to be montonous in n, and when n is itself a power of two, the depth is exactly $\log_2 n$). Thus, the total number of field multiplications in this algorithm is less than $2n \log_2 n$, which justifies the assertion that the complexity is $O(n \log n)$.

Precomputed Tables. Let $w \geq 1$ be an integer; we can precompute powers of g into lookup tables. Namely, we include in the implementation a two-dimensional table G with $G[i][j] = g^{j2^{iw}}$, for $0 \leq j < 2^w$ and $0 \leq iw < n$. This table G is in fact exactly the same as the one used in Bernstein's algorithm, thus necessarily with the same storage cost. Using G:

- The computations of g^{2^b} and g^{2^a} become free, since these values are just specific elements of G (namely, $g^{2^a} = G[\lfloor a/w \rfloor][2^{a \bmod w}]$).
- The computation of g^{2^a-c} becomes the product of about a/w elements of G, and can be computed with at most $\lceil a/w \rceil$ multiplications.
- We can stop the recursion earlier by using a reverse lookup of the current root of 1 in the last sub-table of G, when u is necessarily a member of the roots contained in that sub-table. For instance, if working with the P-224 field with 6-bit tables ($w = 6$), then four recursion layers yield $n = 6$, i.e. the input u at that depth is a 64-th root of 1, and the table $G[15]$ contains exactly all 64 such roots. We can thus simply locate the input in that table.

The reverse lookup can work with only a subset of the input value (as a bit pattern). For instance, if using $q = 2^{224} - 2^{96} + 1$ (the P-224 field) and $w = 6$, them, for an input u which is a 64-th root of 1, we can represent u as a canonical integer in the 0 to $q - 1$ range, and inspect only the 9-bit pattern starting at bit 88 of the value: this 9-bit pattern is enough to disambiguate all 64 possible roots. We can even do slightly better by considering the canonical representation of $2u$ instead: in that case, the 8-bit pattern at offset 24 is enough.

Mutualized Squarings. In DLPpow2, the preparation for the first recursive call implies the computation of u^{2^b}, which normally uses b squarings. The second recursive call receives

$u' = u \cdot g^{2^a - c}$ as parameter, and will itself compute $u'^{2^{b'}}$ when preparing for its own first recursive call. Note that $b' \approx b/2 \approx a/2$, and:

$$\begin{aligned} u'^{2^{b'}} &= u^{2^{b'}} \left(g^{2^a - c}\right)^{2^{b'}} \\ &= u^{2^{b'}} \left(g^{2^{b'}}\right)^{2^a - c} \end{aligned}$$

Since the $b' < b$, we already obtained $u^{2^{b'}}$ for free when computing u^{2^b}. Therefore, to compute $u'^{2^{b'}}$, we can use either b' squarings, or instead reuse the already obtained $u^{2^{b'}}$ and compute $(g^{2^{b'}})^{2^a - c}$, which will need about a/w multiplications using the precomputed tables; if $w \geq 3$, then the latter method is normally faster. This optimization primarily applies at the shallowest level of the recursion, and not at the level immediately below (if that call computes $u'^{2^{b'}}$ with the alternate method, then it iself does *not* get $u'^{2^{b''}}$ "for free"); if n is large then the optimization may also yield some additional gains at depth 2, though the main advantage is at depth 0.

Avoiding the Final Exponentiation. In the square root computation (algorithm 1), we ultimately compute $g^{(2^n - e)/2}$. This exponentiation can be mutualized with the computations in DLPpow2; namely, in addition to returning the exponent e, the function may also return a square root of $g^{-e} = 1/u$. This is applied in DLPpow2ext (algorithm 3). Note that only the rightmost path of the recursion tree applies the modification; the other calls to DLPpow2 are unchanged.

Algorithm 3 DLPpow2ext: Discrete Logarithm and Inverse Square Root

Input: $u, g, s \in GF(q)$ and $n \geq 1$, with $g^{2^{n-1}} = -1$, $u^{2^n} = 1$, and $s^2 = g$ ($s = \perp$ if g is not a square)
Output: (e, t) such that $0 \leq e < 2^n$, $u = g^e$ and $t^2 = 1/u$ (or $(0, \perp)$ if u is not a square)

if $n = 1$ **then**
  if $u = 1$ **then**
    return $(0, 1)$
  else
    return $(1, s)$
$a \leftarrow \lfloor n/2 \rfloor$
$b \leftarrow n - a$
$c \leftarrow$ DLPpow2(u^{2^b}, g^{2^b}, a)
if $s = \perp$ **then**
  if $c \bmod 2 = 1$ **then**
    return $(0, \perp)$
  else
    $k \leftarrow g^{(2^a - c)/2}$
else
  $k \leftarrow s^{2^a - c}$
$(d, z) \leftarrow$ DLPpow2ext$(uk^2, g^{2^a}, g^{2^{a-1}}, b)$
$t \leftarrow kz$
return $(c + (d - 1 \bmod 2^b), t)$

In DLPpow2ext, basis g is in fact equal to the original g raised to the power 2^j for some known integer j, thus $s = \sqrt{g}$ is the original g raised to the power 2^{j-1}. The returned value is correct because the first parameter to DLPpow2ext is the exact same value as was used in the plain DLPpow2, and the obtained z is a square root of $1/(uk^2)$; the value kz is then a square root of $1/u$.

Using DLPpow2ext, the square root extraction itself avoids the final exponentiation (algorithm 4). The exact gain depends on the used tables; notably, the computation of $(\sqrt{g})^{2^a-c}$ might be a worse fit for the table boundaries than g^{2^a-c}, cancelling some of the gains.

Algorithm 4 Square Root Extraction in $GF(q)$ (extended)

Input: $x \in GF(q)$, with $q = 2^n m + 1$ ($m = 1 \bmod 2$, $n \geq 1$)
Output: $y \in GF(q)$ such that $y^2 = x$, or $\perp$ if x is not a square in $GF(q)$

1: **if** $x = 0$ **then**
2: **return** 0
3: $v \leftarrow x^{(m-1)/2}$
4: $w \leftarrow xv$
5: $u \leftarrow wv$
6: $(e, t) \leftarrow$ DLPpow2ext$(u, g, \perp, n)$
7: **if** $t \neq \perp$ **then**
8: **return** wt
9: **else**
10: **return** $\perp$

Shortcuts and Constant-Time Operations. In some practical usage scenarios, the input is non-secret; this is the case for elliptic curve point decompression, when the point to decompress is a public key. For non-secret inputs, some shortcuts may be applied:

- If the input is not a square, then this is detected on the leftmost path of the recursion tree, after only $n-1$ squarings: for the input u, the value $u^{2^{n-1}}$ turns out to be equal to -1, which implies that the discrete logarithm e is odd. The whole square root computation can then be aborted at that point, and return a failure ($\perp$).
- In the precomputed tables, $G[i][0] = 1$ and $G[i][2^{w-1}] = -1$ for all i; we can skip multiplications by 1, and replace multiplications by -1 with significantly cheaper negations. This induces significant savings when w is small (1 or 2).
- More generally, all table lookups are a simple dereferencing at an index-dependent address.

Conversely, when the input is secret, none of these shortcuts may be used. In particular:

- All multiplications must be performed, regardless of whether an operand happens to be equal to 1 or -1. In some cases, a constant-time conditional negation can be used if it can be shown that the operand can only be 1 or -1, for all possible inputs.
- All table lookups must use a constant-time process which reads all entries and combines them using Boolean operations to retain only the value of the correct entry. The cost of a lookup is then proportional to the table size (2^w), which disfavours large tables.

– The case of $x = 0$ cannot be discarded out-of-band; thus, it may happen that zero values occur in the computation. The main effect is that the reverse-lookup (at the deepest recursion level) will get as input a root of 1 *or* a zero, and must be able to gracefully handle the latter case. Note that if $x = 0$, then $w = 0$, so it does not really matter what values DLPpow2ext returns in that case, as long as it does not crash or misbehave in any other detectable way.

These considerations also apply to previous algorithms such as Bernstein's or Sarkar's.

3 Performance

We present in table 1 a synthetic performance evaluation, for various values of n (96 to 512) and w (2 to 8), which were already used as examples in [8] (table 2).

n	w	Bernstein [3]		Sarkar [8]		**This work**	
		S + M	total	S + M	total	S + M	total
96	2	94S + 1178M	1272	182S + 338M	520	310S + 172M	482
	4	92S + 302M	394	170S + 122M	292	191S + 124M	315
	6	90S + 138M	228	146S + 82M	228	142S + 60M	202
	8	88S + 80M	168	100S + 80M	180	142S + 64M	206
128	2	126S + 2082M	2208	239S + 514M	753	390S + 206M	596
	4	124S + 530M	654	228S + 194M	422	233S + 138M	371
	6	122S + 255M	377	235S + 115M	350	233S + 113M	346
	8	120S + 138M	258	136S + 138M	274	188S + 60M	248
256	2	254S + 8252M	8512	519S + 1484M	2003	903S + 464M	1367
	4	252S + 2082M	2334	484S + 514M	998	546S + 320M	866
	6	250S + 948M	1198	469S + 332M	801	546S + 261M	807
	8	248S + 530M	778	448S + 194M	642	461S + 138M	599
512	2	510S + 32898M	33408	991S + 4098M	5089	2056S + 1042M	3098
	4	508S + 8258M	8766	972S + 1538M	2510	1259S + 734M	1993
	6	506S + 3743M	4249	1057S + 955M	2012	1259S + 571M	1830
	8	504S + 2082M	2586	960S + 514M	1474	1086S + 320M	1406

Table 1: Comparison of runtime costs of this work with Bernstein's and Sarkar's algorithms. For each case, the number of multiplications is presented twice, first with squarings and non-squarings separated, then grouped together. Values do *not* include the cost of the initial exponentiation (to the power $(m-1)/2$), which is the same for all algorithms and does not depend on the parameter w. Values for Bernstein's and Sarkar's algorithms are from [8].

Not captured in table 1 is the storage cost. In general, this is $(2^w - 1)n/w$ field elements for all three algorithms (values $G[i][0]$ need not be stored since they are all equal to 1); however, Sarkar's algorithm can somewhat reduce that storage size when w happens not to exactly divide n; in the table examples, this applies to cases $n = 128$, 256 and 512 for $w = 6$.

We see in table 1 that our new algorithm is on par with Sarkar's, and significantly outperforms it when n/w is large. This corresponds to the better asymptotic complexity. Conversely, when w does not exactly divide n, or when n/w is not a power of two, table boundary effects tend to have an adverse effect on our performance; this is especially visible for $n = 96$, where increasing the table sizes from $w = 6$ to $w = 8$ (which triples the storage cost) makes performance strictly worse. Sarkar's algorithm includes a compile-time search for an optimal "split sequence", which tends to better adapt to such conditions. In the algorithm described in this paper, we use a simple split of the input n into $a + b$ with $a = \lfloor n/2 \rfloor$; this is not necessarily optimal, and it is conceivable that for specific values of n and w, some slightly skewed split strategies would yield better performance. This is currently unexplored.

It should be recalled here that in table 1, we only consider field multiplications (including squarings) for the cost. We ignore simpler operations such as conditional selection or inner function calls, and table lookups; the latter, in particular, can have a non-trivial cost if operating on secret input, but also in general through cache effects: if large tables are used, then the square root computation will use some cache resources and thus evict other values, which may indirectly slow down other parts of the application that use the square root primitive. As always, asymptotic complexity, operation counts for specific parameters, primitive microbenchmarks, and in-application performance are different things, and none of them is an accurate predictor of the next one.

4 Conclusion

We presented an evaluation of the application of a divide-and-conquer variant of the Pohlig-Hellman algorithm to the computation of square roots in some "inconvenient" finite fields. This algorithm offers performance which is on par with previsouly best known solutions, but becomes significantly better for large values of n (or small precomputed tables), thanks to a lower asymptotic complexity.

This is probably not an optimal solution. At least from an asymptotical point of view, Sutherland's algorithm[11] promises a lower complexity ($O(n \log n / \log \log n)$ group operations) for solving discrete logarithm in cyclic groups of order 2^n. For specific parameter values, nested strategies that performs carefully selected splits and switch algorithms have been explored in other contexts using such discrete logarithms, e.g. as part of some cryptographic key exchange protocols working on isogeny graphs between supersingular elliptic curves (see [5], section 5). Even if using the algorithm presented here, there are at least some unexplored optimization areas related to the optimal order split and set of precomputed tables to obtain the best performance, especially when w does not divide n, or when n/w is not a power of two.

References

1. L. Adleman, K. Manders and G. Miller, *On taking roots in finite fields*, Proceedings of the 18th Annual Symposium on Foundations of Computer Science, pp. 175-178, 1977.
2. A. Atkin, *Probabilistic primality testing (summary by F. Morain)*, Technical Report 1779, INRIA, 1992, `http://algo.inria.fr/seminars/sem91-92/atkin.pdf`
3. D. Bernstein, *Faster square roots in annoying finite fields*,

https://cr.yp.to/papers.html#sqroot (2001)
4. S. Bowe, J. Grigg and D. Hopwood, *Recursive Proof Composition without a Trusted Setup*, https://eprint.iacr.org/2019/1021
5. C. Costello, D. Jao, P. Longa, M. Naehrig, J. Renes and D. Urbanik, *Efficient compression of SIDH public keys*, Advances in Cryptology - EUROCRYPT 2017, Lecture Notes in Computer Science, vol. 10210, pp. 679-706, 2017.
6. Information Technology Laboratory, *Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters*, National Institute of Standard and Technology, SP 800-186, 2023.
7. S. Pohlig and M. Hellman, *An improved algorithm for computing logarithms over GF(p) and its cryptographic significance*, IEEE Transactions on Information Theory, vol. 24, issue 1, pp. 106-110, 1978.
8. P. Sarkar, *Computing Square Roots Faster than the Tonelli-Shanks/Bernstein Algorithm*, https://eprint.iacr.org/2020/1407
9. D. Shanks, *Five Number Theoretic Algorithms*, Proceedings of the Second Manitoba Conference on Numerical Mathematics, pp. 51-70, 1973.
10. V. Shoup, *A Computational Introduction to Number Theory and Algebra*, Cambridge University Press, 2005, https://shoup.net/ntb/
11. A. Sutherland, *Structure computation and discrete logarithms in finite abelian p-groups*, Mathematics of Computation, vol. 80, issue 273, pp. 477-500.
12. A. Tonelli, *Sulla risoluzione della congruenza* $x^2 \equiv c \pmod{p^\lambda}$, Atti della Reale Accademia dei Lincei, ser. 5, vol. 1, sem. 1, pp. 116-120, 1892.
13. Tablet CBS 01535 (anonymous), Penn Museum, Philadelphia, Achaemenid period (547-331 BC).