# Computing Square Roots Faster than the Tonelli-Shanks/Bernstein Algorithm

Palash Sarkar
Indian Statistical Institute
203, B.T. Road
Kolkata
India 700108
email:palash@isical.ac.in

November 29, 2021

**Abstract**

Let $p$ be a prime such that $p = 1 + 2^n m$, where $n \geq 1$ and $m$ is odd. Given a square $u$ in $\mathbb{Z}_p$ and a non-square $z$ in $\mathbb{Z}_p$, we describe an algorithm to compute a square root of $u$ which requires $\mathfrak{T} + O(n^{3/2})$ operations (i.e., squarings and multiplications), where $\mathfrak{T}$ is the number of operations required to exponentiate an element of $\mathbb{Z}_p$ to the power $(m-1)/2$. This improves upon the Tonelli-Shanks (TS) algorithm which requires $\mathfrak{T} + O(n^2)$ operations. Bernstein had proposed a table look-up based variant of the TS algorithm which requires $\mathfrak{T} + O((n/w)^2)$ operations and $O(2^w n/w)$ storage, where $w$ is a parameter. A table look-up variant of the new algorithm requires $\mathfrak{T} + O((n/w)^{3/2})$ operations and the same storage. In concrete terms, the new algorithm is shown to require significantly fewer operations for particular values of $n$.
**Key Words:** square root, Tonelli-Shanks algorithm, table look-up.
**MSC Codes: 11Y16**

## 1 Introduction

Let $p$ be an odd prime such that $p - 1 = 2^n m$, where $n$ and $m$ are integers with $n \geq 1$ and $m$ odd. Suppose an element $u$ in $\mathbb{Z}_p$ is a square (which can be determined by computing the Legendre symbol $(\frac{u}{p})$). A basic task in many applications is to obtain a square root of $u$ modulo $p$. An example is to find the y-coordinate from the x-coordinate of a point on an elliptic curve which is required for certain applications of elliptic curve cryptography. If $n = 1$ (i.e., $p \equiv 3 \bmod 4$), then $u^{(m+1)/2}$ is a square root of $u$. If $n > 1$ (equivalently, $p \equiv 1 \bmod 4$), there are several approaches to computing a square root of $u$.

The Tonelli-Shanks (TS) algorithm [15, 14] requires $\mathfrak{T} + O(n^2)$ operations[1] (i.e., squarings and multiplications), where $\mathfrak{T}$ is the number of operations required to compute $u^{(m-1)/2}$. The Adleman-Manders-Miller (AMM) algorithm [1] also requires $\mathfrak{T} + O(n^2)$ operations, though in the average case, the Tonelli-Shanks algorithm is faster. A detailed analysis of the average case behaviour of the TS algorithm is provided in [11]. Bernstein [4] proposed a method to speed-up the computation of the TS

[1]The expression $O(n^2)$ assumes that $n$ goes to infinity. This is meaningful, since by Dirichlet's theorem, for every $n$, the set $\{1 + i2^n : i \geq 1\}$ contains infinitely many primes.

algorithm using a pre-computed table containing elements of $\mathbb{Z}_p$. Given a positive integer $w$, Bernstein's method requires $\mathfrak{T} + O((n/w)^2)$ operations and storage of $O(2^w n/w)$ elements.

As mentioned above, for $p \equiv 3 \bmod 4$, a single exponentiation in $\mathbb{Z}_p$ is sufficient to obtain a square root. Atkin [2] showed that a single exponentiation is also sufficient for the case $p \equiv 5 \bmod 8$. Atkin's method was extended to the case $p \equiv 9 \bmod 16$ by Müller in [12], but this requires two exponentiations. Kong et al. [9] provided an improvement to Müller's variant of Atkin's algorithm for $p \equiv 9 \bmod 16$ which requires a single exponentiation for half of the squares, while for the other half two exponentiations are required, so that 1.5 expected exponentiations are required for a random square. Atkin's method was extended to cover the entire case of $p \equiv 1 \bmod 4$ in [13], again requiring two exponentiations plus additional operations. A different approach to square root computation is the Cipolla-Lehmer algorithm [7, 10] which requires working in a quadratic extension of $\mathbb{F}_p$. This approach requires an exponentiation to the power $(p-1)/2$ in the quadratic extension. Müller [12] described a method based on Lucas sequences which is faster than the extensions of Atkin's algorithm in [12, 13] and is also faster than working with quadratic extensions.

In the present state-of-the-art, for $n \geq 3$, depending on the relative values of $n$, $m$ and $\lg p$, among all the known algorithms, either the TS algorithm along with Bernstein's table look-up based variant, or the Lucas sequence based algorithm of Müller [12] is the most efficient. An approximate guide is as follows. Let $c \in [0, 1/2]$ be such that the exponentiation $u^{(m-1)/2}$ requires about $\lg m$ squarings and $c \lg m$ multiplications. If $m$ is very small compared to $\lg p$, then Müller's algorithm is the fastest; if $n^2/(1-c) - ((1+c)/(1-c))n < \lg p$, then the TS algorithm is faster than Müller's algorithm; if $(n/w)^2/(1-c) - ((1+c)/(1-c))n < \lg p$, then Bernstein's algorithm is faster than Müller's algorithm.

Let $z$ be a non-square in $\mathbb{Z}_p$ and set $g = z^m$. The new algorithm starts by computing $v = u^{(m-1)/2}$ as in the TS algorithm. The algorithm has several parameters[2]. Let $k \geq 1$ be a positive integer and let $\ell_0, \ldots, \ell_{k-1}$ be positive integers such that $\ell_0 + \cdots + \ell_{k-1} = n - 1$. Let $x = u \cdot v^2 = u^m$. For $i = 0, \ldots, k-1$, let $x_i = x^{2^{n-1-(\ell_0+\cdots+\ell_i)}}$. In $k$ steps, the goal is to obtain $s_0, s_1, \ldots, s_{k-1}$ and $0 = t_0, t_1, \ldots, t_{k-1}$ such that for $i = 0, \ldots, k-1$, if we define $\alpha_i = x_i \cdot g^{t_i}$, then $\alpha_i \cdot g^{s_i} = 1$. A square root of $u$ is $u^{(m+1)/2} \cdot g^{(t_{k-1}+s_{k-1})/2}$. If $k = 1$, then essentially the TS algorithm is obtained. On the other end, if $k = n-1$ and $\ell_0 = \ell_1 = \cdots = \ell_{k-1} = 1$, then essentially the AMM algorithm is obtained. The complexity of the algorithm (other than the computation of $u^m$) at both these ends are similar where the number of operations is quadratic in $n$. A careful analysis shows that it is possible to choose $k$ and the integers $\ell_0, \ldots, \ell_{k-1}$ such that the number of operations becomes $O(n^{3/2})$.

We propose a table look-up based method to speed up the new algorithm. The idea of using table look-up is an extension of Bernstein's idea of using the table look-up to speed up the TS algorithm. The table look-up based variant of the new algorithm requires $O((n/w)^{3/2})$ operations using a table storing $O(2^w n/w)$ elements.

In practical terms, the new algorithm improves upon the TS algorithm for all $n \geq 3$. We provide concrete examples to illustrate the improvement gain. Similarly, we also provide concrete examples to illustrate the gain in speed of the new look-up table based algorithm in comparison to Bernstein's algorithm [4]. For $n^{3/2}/(1-c) - ((1+c)/(1-c))n < \lg p$, the new algorithm is faster than Müller's algorithm, and for $(n/w)^{3/2}/(1-c) - ((1+c)/(1-c))n < \lg p$, the table look-up based variant of the new algorithm is faster than Müller's algorithm.

[2]These parameters are explained in details later.

## 2 Algorithm

The following result provides the basis of the method for computing square roots.

**Lemma 1** *Let $p$ be an odd prime such that $p-1 = 2^n m$ with $m$ odd and $n \geq 1$. Let $z \in \mathbb{Z}_p^\star$ be a non-square and $g = z^m$. Let $\alpha \in \mathbb{Z}_p^\star$ be such that $\alpha^{2^\ell} = 1$ for some $\ell \in \{0, \ldots, n-1\}$. Then there is an integer $s$ with $0 \leq s < 2^n$ such that $\alpha \cdot g^s = 1$ and $s \equiv 0 \bmod 2^{n-\ell}$. Consequently, it is possible to write $s = q2^{n-\ell}$ with $0 \leq q < 2^\ell$.*

**Proof:** Since $z$ is a non-square, $g^{2^{n-1}} = z^{m2^{n-1}} = z^{(p-1)/2} = -1$. Also, $g^{2^n} = z^{p-1} = 1$. So, the element $g$ is a generator of the cyclic subgroup of $Z_p^\star$ of order $2^n$ and $g^i$, $i = 0, \ldots, 2^n - 1$ are all the distinct roots of the equation $X^{2^n} - 1 = 0$ (over the field $\mathbb{F}_p$). It is given that $\alpha^{2^\ell} = 1$ with $0 \leq \ell \leq n-1$. So, $\alpha^{2^n} = \left(\alpha^{2^\ell}\right)^{2^{n-\ell}} = 1$ which shows that $\alpha$ is a root of $X^{2^n} - 1 = 0$. Consequently, $\alpha$ can be written as a power of $g$, i.e., we may write $\alpha = g^\sigma$, where $0 \leq \sigma < 2^n$. Then $1 = \alpha^{2^\ell} = g^{\sigma 2^\ell}$. Since the order of $g$ is $2^n$, it follows that $\sigma 2^\ell \equiv 0 \bmod 2^n$ and so $\sigma = 0 \bmod 2^{n-\ell}$. Define $s$ as follows: $s = 0$, if $\sigma = 0$; and $s = 2^n - \sigma$, if $\sigma > 0$. Then $s \equiv 0 \bmod 2^{n-\ell}$, $0 \leq s < 2^n$ and $\alpha \cdot g^s = g^\sigma \cdot g^s = 1$. □

Suppose $u \in \mathbb{Z}_p^\star$ be a square and $x = u^m$. Then $x^{2^{n-1}} = u^{(p-1)/2} = 1$ and so, by Lemma 1, there is an even integer $t \geq 0$ such that $x \cdot g^t = 1$. Let $v = u^{(m-1)/2}$. Then $(u \cdot v \cdot g^{t/2})^2 = u \cdot u^m \cdot g^t = u \cdot x \cdot g^t = u$ and so $u \cdot v \cdot g^{t/2}$ is a square root of $u$.

Henceforth, we will assume that the prime $p = 2^n m + 1$, with $n \geq 1$ and $m$ an odd integer. We also assume that a quadratic non-residue $z$ in $\mathbb{Z}_p$ is known and we define $g = z^m$.

**Pre-computed powers of $g$:** Let $g_i = g^{2^i}$, $i = 0, \ldots, n-2$. The powers $(g_i)_{0 \leq i \leq n-2}$ are to be computed and stored. Computing these powers requires $n-2$ squarings. Since the same $g$ can be used for all $u$, we assume that these $n-1$ powers of $g$ are pre-computed and stored.

Given a square $u \in \mathbb{Z}_p^*$, the algorithm for finding a square root of $u$ is described as Function $\mathsf{findSqRoot}(u)$ in Algorithm 1. The algorithm is parameterised by the following choices. Let $k > 0$ be an integer and $\ell_0, \ldots, \ell_{k-1}$ be positive integers such that $\ell_0 + \cdots + \ell_{k-1} = n-1$. The following subroutines are used by the algorithm.

$\mathsf{eval}(\alpha)$: Given $\alpha \in \mathbb{Z}_p^\star$ such that $\alpha^{2^\ell} = 1$ with $0 \leq \ell \leq n-1$, $\mathsf{eval}(\alpha)$ returns $s$ such that $\alpha \cdot g^s = 1$. The existence of such an $s$ is guaranteed by Lemma 1.

$\mathsf{find}(\delta)$: Given $\delta \in \mathbb{Z}_p^\star$, with $\delta \neq 1$ such that $\delta^{2^\ell} = 1$, for some $1 \leq \ell \leq n-1$, $\mathsf{find}(\delta)$ returns $i \geq 0$ such that $\delta^{2^i} = -1$.

Note that Step 5 of the Function $\mathsf{eval}$ in Algorithm 1 makes use of the pre-computed powers of $g$.

We establish the correctness of $\mathsf{findSqRoot}$. To this end, we consider the correctness of the two subroutines $\mathsf{find}$ and $\mathsf{eval}$. The correctness of $\mathsf{find}$ is easy to see. For $\delta \neq 1$, let $i$ be the minimum non-negative integer such that $(\delta^{2^i})^2 = 1$. Then $\delta^{2^i}$ must be equal to $-1$, since in $\mathbb{Z}_p$, the only square roots of 1 are 1 and $-1$. The minimum non-negative $i$ is obtained by repeated squarings of $\delta$.

The correctness of $\mathsf{eval}$ is given by the following result.

**Lemma 2 (Correctness of $\mathsf{eval}$)** *Let $\alpha \in \mathbb{Z}_p^\star$ be such that $\alpha^{2^\ell} = 1$ for some $\ell \in \{0, \ldots, n-1\}$ and let $s$ be the output of $\mathsf{eval}(\alpha)$. Then $\alpha \cdot g^s = 1$.*

**Proof:** In the proof, the step numbers refer to the steps of the Function $\mathsf{eval}$ shown in Algorithm 1.

**Algorithm 1** Algorithm for computing a square root of $u$ modulo an odd prime $p$, where $u$ is a non-zero square in $\mathbb{Z}_p^\star$. In the algorithm, $p-1 = 2^n m$ with $n \geq 1$ and $m$ odd; $g = z^m$ where $z$ is a non-square in $\mathbb{Z}_p^\star$; $k \geq 1$ and $\ell_0, \ldots, \ell_{k-1} > 0$ are such that $\ell_0 + \cdots + \ell_{k-1} = n-1$.

1: **function** findSqRoot($u$)
2: $v \leftarrow u^{(m-1)/2}$;
3: **if** $n = 1$ **then** $y \leftarrow u \cdot v$; **return** $y$; **end if**
4: $x \leftarrow uv^2$; (so that $x = u^m$)
5: let $x_i \leftarrow x^{2^{n-1-(\ell_0+\cdots+\ell_i)}}$, $i = 0, \ldots, k-1$; store $(x_i)_{0 \leq i \leq k-1}$;
6: $s \leftarrow 0$; $t \leftarrow 0$;
7: **for** $i \leftarrow 0$ to $k-1$ **do**
8: $t \leftarrow (s+t)/2^{\ell_i}$; $\gamma \leftarrow g^t$; $\alpha \leftarrow x_i \cdot \gamma$; $s \leftarrow$ eval($\alpha$);
9: **end for**
10: $t \leftarrow s + t$; $\gamma \leftarrow g^{t/2}$; $y \leftarrow u \cdot v \cdot \gamma$;
11: **return** $y$;
12: **end function**.

1: **function** eval($\alpha$)
2: $\delta \leftarrow \alpha$; $s \leftarrow 0$;
3: **while** $\delta \neq 1$ **do**
4: $i \leftarrow$ find($\delta$); $s \leftarrow s + 2^{n-1-i}$;
5: **if** $i > 0$ **then** $\delta \leftarrow \delta \cdot g^{2^{n-1-i}}$; **else** $\delta \leftarrow -\delta$;
6: **end while**
7: **return** $s$;
8: **end function**

1: **function** find($\delta$)
2: $\mu \leftarrow \delta$; $i \leftarrow 0$;
3: **while** $(\mu \neq -1)$ **do** $\mu \leftarrow \mu^2$; $i \leftarrow i+1$; **end while**
4: **return** $i$;
5: **end function**

Let $\delta_0$ be the value assigned to $\delta$ in Step 2 and so $\delta_0 = \alpha$. Let $i_0 = \ell$. In the $j$-th iteration ($j \geq 1$) of the loop in Steps 3 to 6, let the values assigned to $i$ and $\delta$ in Steps 4 and 5 be $i_j$ and $\delta_j$ respectively. From the definition of find we have $\delta_{j-1}^{2^{i_j}} = -1$. Also, from Step 5, $\delta_j = \delta_{j-1} \cdot g^{2^{n-1-i_j}}$.

We first show that the loop in Steps 3 to 6 terminates. We have $\delta_0^{2^{i_0}} = \alpha^{2^\ell} = 1$. For $j \geq 1$, $\delta_j^{2^{i_j}} = \left(\delta_{j-1} \cdot g^{2^{n-1-i_j}}\right)^{2^{i_j}} = \delta_{j-1}^{2^{i_j}} \cdot g^{2^{n-1}} = (-1)(-1) = 1$. So, we have for $j \geq 0$, $\delta_j^{2^{i_j}} = 1$ and $\delta_j^{2^{i_{j+1}}} = -1$ which shows that $i_{j+1} < i_j$. Consequently, $n-1 \geq \ell = i_0 > i_1 > i_2 > \cdots$. This shows that $n-1-i_j > 0$ for $j \geq 1$. The values $i_1, i_2, \ldots$ are returned by find and so these are non-negative integers. Since $\ell = i_0, i_1, i_2, \ldots$ is a strict monotone decreasing sequence of non-negative integers, the loop in Steps 3 to 6 has to terminate.

Suppose that the loop in Steps 3 to 6 in eval runs $r$ times and so $\delta_r = 1$. From the manner in which $s$ is updated, the final value returned at Step 7 is $s = 2^{n-1-i_1} + 2^{n-1-i_2} + \cdots + 2^{n-1-i_r}$. We have $1 = \delta_r = \delta_{r-1} \cdot g^{2^{n-1-i_r}} = \delta_{r-2} \cdot g^{2^{n-1-i_{r-1}}} \cdot g^{2^{n-1-i_r}} = \cdots = \delta_0 \cdot g^s = \alpha \cdot g^s$. □

**Definitions of $t_i$, $\alpha_i$ and $s_i$:** For $i = 0, \ldots, k-1$, let $t_i, \alpha_i$ and $s_i$ be the values assigned to $t$, $\alpha$ and

$s$ respectively in Step 8 of findSqRoot. Let $t_k$ be the value assigned to $t$ in Step 10 of findSqRoot. The quantities $x_0, \ldots, x_{k-1}$ are defined in findSqRoot. From the description of the algorithm, $t_0 = 0$ and the following relations hold.

$$\left.\begin{array}{rcll} t_i & = & (t_{i-1} + s_{i-1})/2^{\ell_i}, & \text{for } i = 1, \ldots, k-1; \\ t_k & = & t_{k-1} + s_{k-1}; & \\ \alpha_i & = & x_i \cdot g^{t_i}, & \text{for } i = 0, \ldots, k-1. \end{array}\right\} \tag{1}$$

**Lemma 3** *For $i = 0, \ldots, k-1$, the following holds.*

1. $\alpha_i^{2^{\ell_i}} = 1$, *for* $i = 0, \ldots, k-1$.
2. $\alpha_i \cdot g^{s_i} = 1$, *for* $i = 0, \ldots, k-1$.

**Proof:** First note that $\alpha_0 = x_0 = x^{2^{n-1-\ell_0}}$ and so $\alpha_0^{2^{\ell_0}} = x^{2^{n-1}} = u^{m2^{n-1}} = u^{(p-1)/2} = 1$, where the last equality holds since $u$ is a square in $\mathbb{Z}_p^\star$. Since $s_0$ is obtained as $\mathsf{eval}(\alpha_0)$, by correctness of eval, it follows that $\alpha_0 \cdot g^{s_0} = 1$.

Assume $1 \leq i \leq k-1$. From (1) we have $2^{\ell_i} t_i = t_{i-1} + s_{i-1}$ and $\alpha_i = x_i \cdot g^{t_i}$. From the definition of $x_i$, we have $x_i = x^{2^{n-1-(\ell_0+\cdots+\ell_i)}}$ and so $x_i^{2^{\ell_i}} = x^{2^{n-1-(\ell_0+\cdots+\ell_{i-1})}} = x_{i-1}$. So, $\alpha_i^{2^{\ell_i}} = x_i^{2^{\ell_i}} \cdot g^{t_i 2^{\ell_i}} = x_{i-1} \cdot g^{t_{i-1}} \cdot g^{s_{i-1}} = \alpha_{i-1} \cdot g^{s_{i-1}} = 1$, where the last equality follows by induction. For $i = 1, \ldots, k-1$, $s_i$ is obtained as $\mathsf{eval}(\alpha_i)$ and by the correctness of eval, it follows that $\alpha_i \cdot g^{s_i} = 1$. □

Using Lemmas 1 and 3, for $i = 0, \ldots, k-1$, we have

$$s_i \;=\; q_i 2^{n-\ell_i}, \text{ with } 0 \leq q_i < 2^{\ell_i}. \tag{2}$$

Note that since $t_0 = 0$, $g^{t_0} = 1$. So, we consider $t_i$ for $i \geq 1$.

**Lemma 4** *For $i = 1, \ldots, k$,*

$$t_i \;=\; \left(q_0 + 2^{\ell_0} q_1 + 2^{\ell_0+\ell_1} q_2 + \cdots + 2^{\ell_0+\cdots+\ell_{i-2}} q_{i-1}\right) 2^{\ell_{i+1}+\cdots+\ell_{k-1}+1}. \tag{3}$$

*Consequently, $t_i \equiv 0 \bmod 2^{\ell_{i+1}+\cdots+\ell_{k-1}+1}$. In particular, both $t_{k-1}$ and $t_k$ are even.*

**Proof:** Recall that $\ell_0 + \cdots + \ell_{k-1} = n-1$. Define $\ell_k = 0$, so that the relation $t_i = (t_{i-1} + s_{i-1})/2^{\ell_i}$ holds for $i = 1, \ldots, k$ (see (1)). We have $t_1 = s_0/2^{\ell_1} = q_0 2^{n-\ell_0}/2^{\ell_1} = q_0 2^{\ell_2+\cdots+\ell_{k-1}+1}$. Assume by induction that the expansion for $t_i$ holds for some $i \in \{1, \ldots, k-1\}$. From (2), we have $s_i = q_i 2^{n-\ell_i}$. Using this and induction, we have

$$\begin{array}{rcl} t_{i+1} & = & \dfrac{1}{2^{\ell_{i+1}}}(t_i + s_i) \\ & = & \dfrac{1}{2^{\ell_{i+1}}}\left(\left(q_0 + 2^{\ell_0} q_1 + 2^{\ell_0+\ell_1} q_2 + \cdots + 2^{\ell_0+\cdots+\ell_{i-2}} q_{i-1}\right) 2^{\ell_{i+1}+\ldots+\ell_{k-1}+1} + q_i 2^{n-\ell_i}\right) \\ & = & \left(q_0 + 2^{\ell_0} q_1 + \cdots + 2^{\ell_0+\cdots+\ell_{i-2}} q_{i-1} + 2^{\ell_0+\cdots+\ell_{i-1}} q_i\right) 2^{\ell_{i+2}+\cdots+\ell_{k-1}+1}. \end{array}$$

□

For $i = 1, \ldots, k$ and $j = 0, \ldots, i-1$, define

$$\kappa_{i,j} \;=\; \ell_0 + \cdots + \ell_{j-1} + \ell_{i+1} + \cdots + \ell_{k-1} + 1. \tag{4}$$

Then the expression for $t_i$ in Lemma 4 can be written as

$$t_i \;=\; q_0 2^{\kappa_{i,0}} + q_1 2^{\kappa_{i,1}} + \cdots + q_{i-1} 2^{\kappa_{i,i-1}}. \tag{5}$$

The expression for $t_i$ in terms of $\kappa_{i,j}$ will be useful in describing the computation of $g^{t_i}$.

Next we establish the correctness of findSqRoot.

**Theorem 1 (Correctness of findSqRoot)** *Let $u \in \mathbb{Z}_p^\star$ be a square and $y$ is the result of* findSqRoot$(u)$. *Then $y^2 = u$.*

**Proof:** Since $u$ is a square, we have $u^{(p-1)/2} = u^{m2^{n-1}} = 1$. If $n = 1$, then $y^2 = (u \cdot v)^2 = u^{m+1} = u^m \cdot u = u$. So, assume that $n > 1$.

From Lemma 4, we have that $t_k$ is even. So the computation $g^{t_k/2}$ in Step 10 of findSqRoot is meaningful.

By Lemma 3, we have $\alpha_{k-1} \cdot g^{s_{k-1}} = 1$. From (1), $\alpha_{k-1} = x_{k-1} \cdot g^{t_{k-1}}$ and by definition, $x_{k-1} = x = u^m$. So, we get $x \cdot g^{t_{k-1}} \cdot g^{s_{k-1}} = u^m \cdot g^{t_{k-1}+s_{k-1}} = 1$. Again, from (1), $t_k = t_{k-1} + s_{k-1}$ and so, $y = u \cdot v \cdot g^{t_k/2} = u^{(m+1)/2} \cdot g^{(t_{k-1}+s_{k-1})/2}$. Therefore $y^2 = u \cdot u^m \cdot g^{t_{k-1}+s_{k-1}} = u$. □

**Remark:** In Appendix A, we describe the relation of findSqRoot to the TS and the AMM algorithms.

## 2.1 Computations of $g^t$ and $g^{t/2}$

Step 8 of findSqRoot requires the computation of $g^{t_i}$, $i = 0, \ldots, k-1$, and Step 10 requires the computation of $g^{t_k/2}$, where the expressions for $t_i$, $i = 1, \ldots, k$ are given by (5).

From (2), we have $q_j < 2^{\ell_j}$, for $j = 0, \ldots, k-1$. So, the binary expansion of $q_j$ can be written as

$$q_j \;=\; q_{j,0} + q_{j,1} 2 + \cdots + q_{j,\ell_i - 1} 2^{\ell_j - 1} \tag{6}$$

where $q_{j,0}, \ldots, q_{j,\ell_j-1} \in \{0, 1\}$. Substituting the expression for $q_j$ given by (6) in (5) we obtain

$$t_i \;=\; \sum_{j=0}^{i-1} \left( q_{j,0} + q_{j,1} 2 + \cdots + q_{j,\ell_i - 1} 2^{\ell_j - 1} \right) 2^{\kappa_{i,j}}.$$

So, for $i = 1, \ldots, k-1$,

$$\begin{aligned} g^{t_i} &= \prod_{j=0}^{i-1} g^{\left(q_{j,0} + q_{j,1} 2 + \cdots + q_{j,\ell_i - 1} 2^{\ell_j - 1}\right) 2^{\kappa_{i,j}}} \\ &= \prod_{j=0}^{i-1} \left( \left( g^{2^{\kappa_{i,j}}} \right)^{q_{j,0}} \left( g^{2^{\kappa_{i,j}+1}} \right)^{q_{j,1}} \cdots \left( g^{2^{\kappa_{i,j}+\ell_j - 1}} \right)^{q_{j,\ell_j - 1}} \right) \\ &= \prod_{j=0}^{i-1} \left( \left( g_{\kappa_{i,j}} \right)^{q_{j,0}} \left( g_{\kappa_{i,j}+1} \right)^{q_{j,1}} \cdots \left( g_{\kappa_{i,j}+\ell_j - 1} \right)^{q_{j,\ell_j - 1}} \right). \end{aligned} \tag{7}$$

Note that the elements $g_{\kappa_{i,j}}, \ldots, g_{\kappa_{i,j}+\ell_j-1}$ are available from the list $(g_i)_{0 \le i \le n-2}$ of pre-computed powers of $g$. As a result, since $q_{j,0}, \ldots, q_{j,\ell_j-1}$ are bits, the computation of

$$\left( \left( g_{\kappa_{i,j}} \right)^{q_{j,0}} \left( g_{\kappa_{i,j}+1} \right)^{q_{j,1}} \cdots \left( g_{\kappa_{i,j}+\ell_j - 1} \right)^{q_{j,\ell_j - 1}} \right) \tag{8}$$

involves multiplications of known elements.

The computation of $g^{t_k/2}$ is similar to (7), with the only difference being that $\kappa_{k,j}$ is to be replaced by $\kappa_{k,j}-1$. Note that $\kappa_{k,k-1}+\ell_{k-1}-1 = n-2$ and the corresponding power $g^{2^{n-2}}$ has been pre-computed. This is the highest power of $g$ that is required in the computations in Step 8 and 10 of findSqRoot.

### 2.2 Choice of $\ell_0, \ldots, \ell_{k-1}$

In Algorithm findSqRoot, the only required condition on $\ell_0, \ldots, \ell_{k-1}$ is $\ell_0 + \cdots + \ell_{k-1} = n-1$. For any choice of $\ell_0, \ldots, \ell_{k-1}$ satisfying this condition, the algorithm correctly computes a square root of $u$. In practice, a particular choice of values for $\ell_0, \ldots, \ell_{k-1}$ will be required. We mention one method of choosing these parameters so that they are more or less equal. Given $n$ and $k$, let $\ell$, $k_1$, $k_2$ and $\ell_0, \ldots, \ell_{k-1}$ be chosen as follows.

$$\left.\begin{array}{rcll} n-1 &=& (\ell-1)k + k_2, & 1 \le k_2 \le k; \\ k_1 &=& k - k_2; & \\ \ell_i &=& \ell - 1, & \text{for } i = 0, \ldots, k_1 - 1; \\ \ell_i &=& \ell, & \text{for } i = k_1, \ldots, k-1. \end{array}\right\} \tag{9}$$

The above ensures that the first $k_1$ of the $\ell_i$'s are equal to $\ell - 1$ and the last $k_2$ of the $\ell_i$'s are equal to $\ell$. If $k$ divides $n-1$, then all the $\ell_i$'s are equal to $\ell$.

### 2.3 Complexity

The dominant operations in the algorithm are squarings and multiplications. So, to determine the time complexity, we determine the numbers of squarings and multiplications. By [S] (resp. [M]) we will denote a squaring (resp. multiplication).

Step 2 of findSqRoot performs the computation $v \leftarrow u^{(m-1)/2}$. This constitutes the first phase of the algorithm. Let $\mathfrak{T}$ denote the total number of multiplications and squarings required for this computation. We analyse the number of squarings and multiplications required by the rest of the algorithm.

Suppose $\ell$ is such that $\delta^{2^\ell} = 1$ and $\delta \neq 1$. The call find($\delta$) returns $i \ge 0$ such that $\delta^{2^i} = -1$. The value of $i$ is determined by repeated squarings. So, find($\delta$) requires at most $(\ell-1)$[S] to determine $i$.

Consider the call eval($\alpha$) where $\alpha^{2^\ell} = 1$. From Lemma 2, the values of $i$ in the while loop inside eval form a strict monotone decreasing sequence of non-negative integers whose largest element is at most $\ell$. These values of $i$ are returned by the calls find. So, the maximum number of squarings required by all the calls to find in eval($\alpha$) is $(\ell-1) + (\ell-2) + \cdots + 1 = \ell(\ell-1)/2$. In addition, eval requires $r$[M] (for updating $\delta$), where $r$ is the number of times the while loop in Steps 3 to 6 of eval executes. We have $r \le \ell$ and we use the upper bound for the worst case analysis.

In the worst case, eval($\alpha_0$), ..., eval($\alpha_{k-1}$) requires

$$\sum_{i=0}^{k-1} \frac{\ell_i(\ell_i-1)}{2}[\mathrm{S}] + (\ell_0 + \cdots + \ell_{k-1})[\mathrm{M}] \;=\; \sum_{i=0}^{k-1} \frac{\ell_i(\ell_i-1)}{2}[\mathrm{S}] + (n-1)[\mathrm{M}]. \tag{10}$$

The computation given in (8) requires at most $(\ell_j - 1)$[M]. Consequently, the computation of $g^{t_i}$, for $i \in \{1, \ldots, k-1\}$ in (7) requires at most $((\ell_0-1) + (\ell_1-1) + \cdots + (\ell_{i-1}-1) + (i-1))$[M]. The term $(i-1)$[M] arises due to the requirement of multiplying the $i$ factors in the product over $j$ from 0

to $i-1$. Similarly, at most $((\ell_0-1)+(\ell_1-1)+\cdots+(\ell_{k-1}-1)+(k-1))$[M] are required to compute $g^{t_k/2}$. So, the number of multiplications in the computations of $g^{t_1},\ldots,g^{t_{k-1}}$ and $g^{t_k/2}$ is at most

$$\begin{aligned}&\sum_{i=1}^{k}((\ell_0-1)+(\ell_1-1)+\cdots+(\ell_{i-1}-1))+k(k-1)/2\\ &\quad=\quad k(\ell_0-1)+(k-1)(\ell_1-1)+\cdots+2(\ell_{k-2}-1)+(\ell_{k-1}-1)+k(k-1)/2. \qquad (11)\end{aligned}$$

Other than $v=u^{(m-1)/2}$, eval, $g^t$ and $g^{t/2}$ the rest of the computation of findSqRoot and the associated numbers of squarings and multiplications are as follows.

$$\left.\begin{array}{rl}\text{Computing } x \text{ as } u\cdot v^2\text{:} & 1\text{[S]}+1\text{[M]},\\ \text{Computing } (x_i)_{0\le i\le k-2}\text{:} & (n-1-\ell_0)\text{[S]},\\ \text{Multiplications in Step 8:} & (k-1)\text{[M]},\\ \text{Multiplications in Step 10:} & 2\text{[M]}.\end{array}\right\} \qquad (12)$$

Note that in Step 8, for $i=0$, $\gamma=g^{t_0}=1$ and $\alpha_1=x_1\cdot\gamma$, so that there is no multiplication, which is the reason that there are a total of $k-1$ multiplications, instead of $k$ multiplications in Step 8.

In view of the above analysis, we have the following result.

**Theorem 2** *The maximum numbers of multiplications and squarings required by* findSqRoot *other than those considered in* $\mathfrak{T}$ *(for the computation of* $v$*) are as follows.*

$$\textit{Squarings:}\qquad (n-\ell_0)+\sum_{i=0}^{k-1}\frac{\ell_i(\ell_i-1)}{2}, \qquad (13)$$

$$\textit{Multiplications:}\qquad n+k+1+\frac{k(k-1)}{2}+\sum_{i=0}^{k-1}(k-i)(\ell_i-1). \qquad (14)$$

Suppose $k=1$ and $\ell_0=n-1$. Then the maximum number of operations required is $((n+1)(n-2)/2)$[S]+$(3n-1)$[M]. At the other end, if $k=n-1$ and $\ell_0=\cdots=\ell_{k-1}=1$, then the maximum number of operations required is $2(n-2)$[S]+$(2n+1+(n-1)(n-2)/2)$[M]. The number of operations required by both of these options is quadratic in $n$.

It is possible to choose values of $k$ and $\ell_0,\ldots,\ell_{k-1}$ so as to balance the expressions in (13) and (14). This leads to a better complexity as stated in the following result.

**Theorem 3** *It is possible to choose the parameters* $k$ *and* $\ell_0,\ldots,\ell_{k-1}$ *such that the number of multiplications and squarings required by* findSqRoot *is* $\mathfrak{T}+O(n^{3/2})$*, where* $\mathfrak{T}$ *is the number of multiplications and squarings required to compute* $u^{(m-1)/2}$*.*

**Proof:** We may balance the expressions in (13) and (14) by choosing $k$ and $\ell_0,\ldots,\ell_{k-1}$ to be about $\sqrt{n}$. More precisely, choose $k=\lfloor\sqrt{n-1}\rfloor$. Using $n$ and $k$, choose $\ell_0,\ldots,\ell_{k-1}$ as in (9). Note that with these choices, all of the quantities $k$ and $\ell_0,\ldots,\ell_{k-1}$ are $O(n^{1/2})$. Consequently, both (13) and (14) are $O(n^{3/2})$. This leads to the time complexity of findSqRoot to be $\mathfrak{T}+O(n^{3/2})$. □

For practical assessment of the efficacy of the algorithm, the average case complexity is more important. We consider the average case complexity of findSqRoot. This requires considering the average case complexity of eval and the average case complexity of exponentiating $g$.

The average case analysis of eval is based on the average case analysis of the Tonelli-Shanks algorithm by Lindhurst [11]. In the average case analysis, the probability is over a random $u$ and also a random generator $g$ of the cyclic subgroup of order $2^n$ of $\mathbb{Z}_n^\star$. From the randomness of $u$, it follows that the inputs $\alpha$ to eval are also random.

**Lemma 5** *Let $\alpha \in \mathbb{Z}_p^\star$ be such that $\alpha^{2^\ell} = 1$ for some $\ell \in \{0, \ldots, n-1\}$. Then* $\mathsf{eval}(\alpha)$ *requires $(\ell(\ell-1)/4)$[S]+$((\ell-1)/2)$[M] operations in the average case, where the probability is considered over a random choice of $\alpha$ and a random choice of $g$.*

**Proof:** Since $\alpha^{2^\ell} = 1$, it follows that $\alpha$ is a random element of a subgroup of order $2^\ell$.

Suppose the loop in Steps 3 to 6 of eval runs for $r$ iterations. Let $\delta_0 = \alpha$ and $i_0 = \ell$ and for $j = 1, \ldots, r$, let $\delta_j$ and $i_j$ be defined as in the proof of Lemma 2. We have $\ell = i_0 > i_1 > i_2 > \cdots > i_r \geq 0$. Also, for $j = 0, \ldots, r-1$, we have the following relations from the proof of Lemma 2: $\delta_j^{2^{i_j}} = 1$, $\delta_{j+1} = \delta_j \cdot g^{2^{n-1-i_j}}$ and $\delta_r = 1$. For $j = 0, \ldots, r-1$, the probability that the random element $\delta_j$ of a subgroup of order $2^{i_j}$ is transformed (by multiplication with $g^{2^{n-1-i_j}}$) to a random element $\delta_{j+1}$ of a subgroup of order $2^{i_{j+1}}$ is $2^{i_{j+1}-i_j}$. Also, the probability that $\delta_r = 1$ is obtained by transforming (on multiplying by $g^{2^{n-1-i_{r-1}}}$)) the random element $\delta_{r-1}$ is $2^{-i_{r-1}}$. So, the probability of the sequence $i_1 > i_2 > \cdots > i_r$ is

$$\frac{1}{2^{\ell-i_1}} \times \frac{1}{2^{i_1-i_2}} \times \cdots \times \frac{1}{2^{i_{r-2}-i_{r-1}}} \cdot \frac{1}{2^{i_{r-1}}} \quad = \quad \frac{1}{2^\ell}.$$

So, each of the possible $2^\ell$ sequences $\ell > i_1 > i_2 > \cdots > i_r \geq 0$ occurs with the same probability $2^{-\ell}$.

Let us first consider the average number of multiplications in Step 5 of eval. Given a sequence $\ell - 1 \geq i_1 > i_2 > \ldots > i_r \geq 0$, the number of multiplications in Step 5 is $r$ or $r-1$ according as $i_r > 0$ or $i_r = 0$. The number of sequences with $i_r = 0$ is $\binom{\ell-1}{r-1}$ and the number of sequences with $i_r > 0$ is $\binom{\ell-1}{r}$. So, the expected number of multiplications in Step 5 is

$$\begin{aligned} \frac{1}{2^\ell} \times \sum_{\imath=0}^{\ell} \left( \imath \binom{\ell-1}{\imath} + (\imath-1)\binom{\ell-1}{\imath-1} \right) &= \frac{1}{2^\ell} \times \left( \sum_{\imath=0}^{\ell} \imath \binom{\ell}{\imath} - \sum_{\imath=0}^{\ell} \binom{\ell-1}{\imath-1} \right) \\ &= \frac{1}{2^\ell} \times \left( \ell 2^{\ell-1} - 2^{\ell-1} \right) = \frac{\ell-1}{2}. \end{aligned}$$

Given a sequence $i_1 > i_2 > \cdots > i_r$, the number of squarings done in the calls to find is $i_1 + i_2 + \cdots + i_r$. So, the expected number of squarings is

$$\frac{1}{2^\ell} \times C_\ell \tag{15}$$

where $C_\ell = \sum_{r=0}^{\ell} (i_1 + i_2 + \cdots + i_r)$. Note that $C_1 = 0$. We obtain a recurrence relation for $C_\ell$. Consider $C_{\ell+1}$. A total of $2^{\ell+1}$ sequences are considered in $C_{\ell+1}$. Of these, $2^\ell$ sequences start with $i_1 = \ell$ and the other $2^\ell$ sequence start with $i_1 < \ell$. The contribution to $C_{\ell+1}$ from the second class of sequences is $C_\ell$, while the contribution to $C_{\ell+1}$ from the first class of sequences is $C_\ell + 2^\ell \ell$. So, we have the relation

$C_{\ell+1} = 2C_\ell + 2^\ell \ell$. We may write

$$\begin{aligned} C_\ell &= 2C_{\ell-1} + 2^{\ell-1}(\ell-1) \\ &= 2\left(2C_{\ell-2} + 2^{\ell-2}(\ell-2)\right) + 2^{\ell-1}(\ell-1) \\ &= 2^2 C_{\ell-2} + 2^{\ell-1}((\ell-1)+(\ell-2)) \\ &\cdot \quad \cdots \\ &= 2^{\ell-1}((\ell-1)+(\ell-2)+\cdots+1) \quad (\text{using } C_1 = 0) \\ &= 2^{\ell-1}\frac{\ell(\ell-1)}{2}. \end{aligned}$$

Combining with (15), the expected number of squarings is obtained to be $\ell(\ell-1)/4$.

□

**Theorem 4** *The average numbers of multiplications and squarings required by* findSqRoot *other than those considered in* $\mathfrak{T}$ *(for the computation of* $v$*) is as follows.*

$$\textit{Squarings:} \qquad (n-\ell_0) + \sum_{i=0}^{k-1} \frac{\ell_i(\ell_i-1)}{4}, \tag{16}$$

$$\textit{Multiplications:} \qquad \frac{n+k+3}{2} + \frac{k(k-1)}{2} + \frac{1}{2}\left(\sum_{i=0}^{k-1}(k-i)(\ell_i-1)\right). \tag{17}$$

**Proof:** The operations counted in (12) remain the same for both worst case and average case. The average case number of operations required by a call to eval($\alpha$), where $\alpha^{2^\ell} = 1$ is given by Lemma 5. Algorithm findSqRoot makes the calls eval($\alpha_0$), ..., eval($\alpha_{k-1}$), where $\alpha_i^{\ell_i} = 1$, for $i = 0, \ldots, k-1$. So, the average case number of operations required by all the calls to eval is

$$\left(\sum_{i=0}^{k-1} \frac{\ell_i(\ell_i-1)}{4}\right)[\mathrm{S}] + \left(\sum_{i=0}^{k-1} \frac{\ell_i - 1}{2}\right)[\mathrm{M}] \quad = \quad \left(\sum_{i=0}^{k-1} \frac{\ell_i(\ell_i-1)}{4}\right)[\mathrm{S}] + \left(\frac{n-k-1}{2}\right)[\mathrm{M}].$$

We next consider the average number of operations required in the computations of $g^{t_1}, \ldots, g^{t_{k-1}}$ and $g^{t_k/2}$. The computation of $g^{t_i}$ is shown in (7) and (8). On an average, about half of the bits $q_{j,0}, \ldots, q_{j,\ell_j-1}$ will be 0 and so, in the average case, the computation in (8) requires $((\ell_j-1)/2)$[M], so that on an average, the computation of $g^{t_i}$, $i = 1, \ldots, k-1$ requires $(((\ell_0-1)+(\ell_1-1)+\cdots+(\ell_{i-1}-1))/2+(i-1))$[M]. Similarly, on an average, the computation of $g^{t_k/2}$ requires $(((\ell_0-1)+(\ell_1-1)+\cdots+(\ell_{k-1}-1))/2+(k-1))$[M]. So, on an average, the number of multiplications required for the computations of $g^{t_1}, \ldots, g^{t_{k-1}}$ and $g^{t_k/2}$ is

$$\frac{1}{2}\left(\sum_{i=0}^{k-1}(k-i)(\ell_i-1)\right) + \frac{k(k-1)}{2}.$$

Putting togther the different counts provides the average number of operations stated in the theorem. □

**Cost of Exponentiation:** The exponentiation $u^{(m-1)/2}$ requires about $\lg m$ squarings and some multiplications. If the exponentiation is done bit-by-bit, then the number of multiplications is equal to the weight of the binary representation of $(m-1)/2$. If the binary representation of $(m-1)/2$ is sparse, then this is quite efficient. On the other hand, if the binary representation is not so sparse, there are known methods for reducing the number of multiplications. We refer to [5] for a survey. We assume that the exponentiation $u^{(m-1)/2}$ requires about $\lg m$ squarings and $c \lg m$ multiplications, for some constant $c \in (0, 1/2]$[3]. So, the total number of operations (i.e., counting both multiplications and squarings) required for the exponentiation $u^{(m-1)/2}$ is about $(1+c)\lg m$.

### 2.4 Concrete Comparison

Both the Tonelli-Shanks algorithm and the new algorithm require the exponentiation $u^{(m-1)/2}$. So, the concrete comparison in this section does not include the number of operations required for this exponentiation.

From [11], the average complexity of the Tonelli-Shanks algorithm (including the initialisation consisting of two multiplications) is $2 + (n^2 + 7n - 12)/4 + 2^{1-n}$ operations (i.e., counting both squarings and multiplications). The division of this total into the number of squarings and the number of multiplications is not provided in [11].

In Table 1, we provide a comparison of the average case complexity of the TS algorithm with that of the new algorithm. The column #TS provides the average number of operations required by the TS algorithm determined according to the above mentioned formula. For the new algorithm, the value of $k$, the break up of the operations into squarings and multiplications and also the total number of operations are provided in the table. Given $n$ and $k$, the values of $\ell_0, \ldots, \ell_{k-1}$ are determined as given by (9). The numbers for the new algorithm given in Table 1 were generated by writing a simple Python program to evaluate the expressions given in (16) and (17). The value of $k$ was varied from 1 to $\lceil \sqrt{n} \rceil$ and the value of $k$ for which the sum of the expressions in (16) and (17) is minimised is reported in the table.

In most implementations, a squaring will be faster than a multiplication. So, along with the total number of operations, the proportion of the number of squarings and the number of multiplications also needs to be taken into consideration. For example, for $n = 32$, the minimum total operation count is 134.5=66.5[S]+68[M] which is obtained for $k = 5$; for $k = 4$, the total operation count is 136=77.5[S]+58.5[M]. Even though the total operation count for $k = 4$ is higher than that for $k = 5$, due to the higher ratio of squarings to multiplications, the choice of $k = 4$ will be faster than the choice of $k = 5$ on almost all systems. So, the figures in Table 1 should be seen only as comparing the TS algorithm with the new algorithm. For the actual choice of $k$ in the new algorithm, the expressions in (16) and (17) will require a closer look.

For $n = 2$, the numbers of operations required by the TS algorithm and the new algorithm are the same. The average case complexity of the new algorithm is lower than the average case complexity of the TS algorithm for $n \geq 3$. The gap grows as $n$ increases.

**Complete cost comparison:** The complete comparison of the Tonelli-Shanks algorithm with the new algorithm has to include the number of operations required for the exponentiation $u^{(m-1)/2}$. Since both algorithms perform this exponentiation, the difference in the number of operations between the two

[3] Using 2-bit window based exponentiation results in $c$ being about 1/2.

Table 1: Comparison of the average case number of operations required by the new algorithm and the TS algorithm.

| $n$ | #TS | new algorithm | | |
|---|---|---|---|---|
| | | $k$ | ops | #tot |
| 16 | 91 | 3 | 26.0[S]+26.0[M] | 52.0 |
| 32 | 311 | 5 | 66.5[S]+68.0[M] | 134.5 |
| 48 | 659 | 6 | 121.5[S]+114.0[M] | 235.5 |
| 64 | 1135 | 7 | 181.0[S]+170.0[M] | 351.0 |
| 80 | 1739 | 8 | 246.5[S]+231.5[M] | 478.0 |
| 96 | 2471 | 9 | 313.5[S]+300.0[M] | 613.5 |

algorithms is not affected by considering the exponentiation. On the other hand, if one is interested in the ratio of the savings in the number of operations to the total number of operations required by the Tonelli-Shanks algorithm, then the number of operations required by the exponentiation becomes important. As discussed earlier, the number of operations required for performing $u^{(m-1)/2}$ is about $\lg m$ squarings and $c \lg m$ multiplications for some $c \in (0, 1/2]$. There are two difficulties for providing concrete estimates of the total number of operations required by the new algorithm (which includes the number of operations required for performing $u^{(m-1)/2}$) as a function of $n$. For one thing, the value of $\lg m$ can vary independently of $n$. Second, as discussed above, the value of $c$ would depend on the sparsity of the binary representation of $(m-1)/2$ and also on the actual exponentiation algorithm that is chosen. Instead of a general discussion, we discuss an example. Consider the prime $p = 2^{224} - 2^{96} + 1$ and so $n = 96$ and $m = 2^{128} - 1$. The exponentiation $u^{(m-1)/2}$ requires 126[S]+10[M] (see [4]). Adding this to the estimates in Table 1 for $n = 96$, the average number of operations (counting both multiplications and squarings) required by the Tonelli-Shanks algorithm is $2471 + 136 = 2607$, while the new algorithm requires 439.5[S]+310[M] for a total of 749.5 operations which is about one-fourth of the number of operations required by the Tonelli-Shanks algorithm.

## 3 Table Look-Up

Given $n$, $k$ and the values $\ell_0, \ldots, \ell_{k-1}$, a parameter $w$ is chosen such that $1 \leq w \leq \max(\ell_0, \ldots, \ell_{k-1})$.

For $i = 0, \ldots, k-1$, let $\tau_i = \lceil \ell_i / w \rceil - 1$ and $\tau^\star = \max\{\tau_0, \ldots, \tau_{k-1}\}$. The following tables are to be pre-computed and stored. These tables are all based on $g = z^m$ and need to be computed only once.

$\mathsf{Tab}_1$: This table consists of triplets of the form $(\nu, i, g^{\nu 2^{n-iw}})$, for $\nu \in \{1, \ldots, 2^w - 1\}$ and $i = 2, \ldots, \tau^\star$, indexed on the first two components.

$\mathsf{Tab}_2$: This table consists of triplets of the form $(\nu, i, g^{\nu 2^{iw}})$ for $\nu \in \{1, \ldots, 2^w - 1\}$ and $i = 0, \ldots, \lceil n/w \rceil - 1$, indexed on the first two components.

$\mathsf{Tab}_3$: This table consists of pairs of the form $(h^{-\nu}, \nu)$, for $\nu \in \{1, \ldots, 2^w - 1\}$ and $h = g^{2^{n-w}}$, indexed on the first component.

$\mathsf{Tab}_1$ stores $(2^w - 1)(\tau^\star - 1)$ tuples; $\mathsf{Tab}_2$ stores $(2^w - 1)(\lceil n/w \rceil)$ tuples; and $\mathsf{Tab}_3$ stores $2^w - 1$ tuples. So, the total storage requirement is $(2^w - 1)(\tau^\star + \lceil n/w \rceil)$ tuples. If $w$ divides $n$, then $\mathsf{Tab}_1$ becomes a part of $\mathsf{Tab}_2$ and the total storage requirement is $(2^w - 1)(n/w + 1)$ tuples.

Based on the parameter $w$, we define some quantities which will be required in the next two sections.

1. For $i = 0, \ldots, k-1$, let $\mathfrak{s}_i$ and $\mathfrak{r}_i$ be such that

$$n - \ell_i \;=\; \mathfrak{s}_i w + \mathfrak{r}_i, \tag{18}$$

where $0 \leq \mathfrak{r}_i < w$.

2. From (4), recall the definition of $\kappa_{i,j}$, $i = 1, \ldots, k$ and $j = 0, \ldots, i-1$. We define $\zeta_{i,j}$ and $\rho_{i,j}$ to be such that

$$\left.\begin{array}{rcll} \kappa_{i,j} & = & \zeta_{i,j} w + \rho_{i,j} & i = 1, \ldots, k-1,\ j = 0, \ldots, i-1; \\ \kappa_{k,j} - 1 & = & \zeta_{k,j} w + \rho_{k,j} & j = 0, \ldots, k-1; \end{array}\right\} \tag{19}$$

where $0 \leq \rho_{i,j} < w$ for $i = 1, \ldots, k$ and $j = 0, \ldots, i-1$.

### 3.1 Computation of eval

Algorithm findSqRoot makes a total of $k$ calls to eval on the $k$ inputs $\alpha_0, \ldots, \alpha_{k-1}$. The call eval$(\alpha_i)$ returns $s_i$, where from Lemma 3, $\alpha_i \cdot g^{s_i} = 1$, and from (2), $s_i = q_i 2^{n-\ell_i}$ with $0 \leq q_i < 2^{\ell_i}$ for $i = 0, \ldots, k-1$. The description of eval is provided in Algorithm 1. Using the previously mentioned tables, we describe a different method of implementing eval.

From the relation $\alpha_i \cdot g^{s_i} = 1$, we have $\alpha_i = g^{-s_i}$. So, the task is to find $s_i$, given $\alpha_i = g^{-s_i}$. Since $s_i = q_i 2^{n-\ell_i}$, it is sufficient to find $q_i$.

Suppose, we write $q_i$ to base $2^w$ in the following form

$$q_i \;=\; q_{i,\tau_i}^{(w)} 2^{\ell_i - w} + q_{i,\tau_i - 1}^{(w)} 2^{\ell_i - 2w} + \cdots + q_{i,1}^{(w)} 2^{\ell_i - \tau_i w} + q_{i,0}^{(w)}, \tag{20}$$

Here $q_{i,1}^{(w)}, \ldots, q_{i,\tau_i}^{(w)} \in \{0, \ldots, 2^w - 1\}$ and $q_{i,0}^{(w)} \in \{0, \ldots, 2^{\ell_i - \tau_i w}\}$. In other words, $q_{i,1}^{(w)}, \ldots, q_{i,\tau_i}^{(w)}$ are $w$-bit integers while $q_{i,0}^{(w)}$ is the left-over $(\ell_i - \tau_i w)$-bit integer. If $w$ divides $\ell_i$, then all of the quantities $q_{i,0}^{(w)}, q_{i,1}^{(w)}, \ldots, q_{i,\tau_i}^{(w)}$ are $w$-bit intgers. So, computing $q_i$ amounts to computing the values $q_{i,0}^{(w)}, q_{i,1}^{(w)}, \ldots, q_{i,\tau_i}^{(w)}$. We describe how this is done.

The first step is to compute and store the powers $\alpha_i^{2^\imath}$ for $\imath = 0, w, 2w, \ldots, \tau_i w$. This requires a total of $\tau_i w$ squarings.

Next consider $s_i = q_i 2^{n-\ell_i}$ written out in the following manner.

$$\begin{array}{rcl} s_i & = & \Big( q_{i,\tau_i}^{(w)} 2^{\ell_i - w} + q_{i,\tau_i - 1}^{(w)} 2^{\ell_i - 2w} + \cdots + q_{i,j+1}^{(w)} 2^{\ell_i - (\tau_i - j)w} + q_{i,j}^{(w)} 2^{\ell_i - (\tau_i - j + 1)w} \\ & & \quad + q_{i,j-1}^{(w)} 2^{\ell_i - (\tau_i - j + 2)w} + \cdots + q_{i,1}^{(w)} 2^{\ell_i - \tau_i w} + q_{i,0}^{(w)} \Big) 2^{n - \ell_i}. \end{array} \tag{21}$$

For $j = 0, \ldots, \tau_i$, multiplying both sides of (21) by $2^{w(\tau_i - j)}$ we have,

$$\begin{array}{rcll} & & \Big( -s_i + \Big( q_{i,j-1}^{(w)} 2^{\ell_i - (\tau_i - j + 2)w} + q_{i,j-2}^{(w)} 2^{\ell_i - (\tau_i - j + 3)w} + \cdots + q_{i,1}^{(w)} 2^{\ell_i - \tau_i w} + q_{i,0}^{(w)} \Big) 2^{n - \ell_i} \Big) 2^{(\tau_i - j)w} & (22) \\ & = & -\Big( q_{i,\tau_i}^{(w)} 2^{\ell_i - w} + q_{\tau_i - 1}^{(w)} 2^{\ell_i - 2w} + \cdots + q_{i,j+1}^{(w)} 2^{\ell_i - (\tau_i - j)w} + q_{i,j}^{(w)} 2^{\ell_i - (\tau_i - j + 1)w} \Big) 2^{n - \ell_i} 2^{(\tau_i - j)w} & \\ & = & -2^n \Big( q_{i,\tau_i}^{(w)} 2^{(\tau_i - j - 1)w} + q_{i,\tau_i - 1}^{(w)} 2^{(\tau_i - j - 2)w} + \cdots + q_{i,j+1}^{(w)} \Big) - q_{i,j}^{(w)} 2^{n-w} & \\ & = & -2^n \chi - q_{i,j}^{(w)} 2^{n-w}. & (23) \end{array}$$

Here $\chi = q_{i,\tau_i}^{(w)} 2^{(\tau_i - j - 1)w} + q_{i,\tau_i - 1}^{(w)} 2^{(\tau_i - j - 2)w} + \cdots + q_{i,j+1}^{(w)}$ is a positive integer.

If we raise both (22) and (23) to the power of $g$, then since $g^{2^n} = 1$, the expression corresponding to (23) becomes $g^{-2^{n-w} q_{i,j}^{(w)}} = h^{-q_{i,j}^{(w)}}$. The expression corresponding to (22) becomes

$$\begin{aligned}
&\left(g^{-s_i}\right)^{2^{(\tau_i - j)w}} \cdot \left(g^{q_{i,j-1}^{(w)} 2^{n-2w}}\right) \cdot \left(g^{q_{i,j-2}^{(w)} 2^{n-3w}}\right) \cdots \left(g^{q_{i,1}^{(w)} 2^{n-jw}}\right) \cdot \left(g^{q_{i,0}^{(w)} 2^{n-\ell_i + (\tau_i - j)w}}\right) \\
&= \quad (\alpha_i)^{2^{(\tau_i - j)w}} \cdot \left(g^{q_{i,j-1}^{(w)} 2^{n-2w}}\right) \cdot \left(g^{q_{i,j-2}^{(w)} 2^{n-3w}}\right) \cdots \left(g^{q_{i,1}^{(w)} 2^{n-jw}}\right) \cdot \left(g^{q_{i,0}^{(w)} 2^{n-\ell_i + (\tau_i - j)w}}\right) \\
&= \quad (\alpha_i)^{2^{(\tau_i - j)w}} \cdot \left(g^{q_{i,j-1}^{(w)} 2^{n-2w}}\right) \cdot \left(g^{q_{i,j-2}^{(w)} 2^{n-3w}}\right) \cdots \left(g^{q_{i,1}^{(w)} 2^{n-jw}}\right) \cdot \left(g^{q_{i,0}^{(w)} 2^{(\mathfrak{s}_i + \tau_i - j)w}}\right)^{2^{\mathfrak{r}_i}}. \qquad (24)
\end{aligned}$$

The relation given by (18) has been used in obtaining the last step.

So, we have $h^{-q_{i,j}^{(w)}}$ equal to the expression given by (24). There are two cases.

*Case* $j = 0$*:* In this case, (24) becomes $(\alpha_i)^{2^{\tau_i w}}$ which is obtained from the computed powers of $\alpha_i$. So $h^{-q_{i,0}^{(w)}} = (\alpha_i)^{2^{\tau_i w}}$.

*Case* $1 \leq j \leq \tau_i$*:* For this case, the terms of (24) may be obtained as follows.

1. The first term of (24) is obtained from the computed powers of $\alpha_i$.
2. The last term of (24) is obtained as follows. Suppose $q_{i,0}^{(w)}$ is known. Then the quantity within the parenthesis is obtained from $\mathsf{Tab}_2$; this quantity is squared $\mathfrak{r}_i$ times to obtain the last term of (24).
3. Suppose $q_{i,1}^{(w)}, \ldots, q_{i,j-1}^{(w)}$ are known. Then the middle $j - 1$ terms of (24) are obtained from $\mathsf{Tab}_1$.

So, for $1 \leq j \leq \tau_i$, given the values of $q_{i,0}^{(w)}, q_{i,1}^{(w)}, \ldots, q_{i,j-1}^{(w)}$, the whole expression in (24) can be computed using $j$ multiplications and $\mathfrak{r}_i$ squarings.

In view of the above two cases, for $0 \leq j \leq \tau_i$, the value of $h^{-q_{i,j}^{(w)}}$ can be obtained. Using $\mathsf{Tab}_3$, from $h^{-q_{i,j}^{(w)}}$ it is possible to find $q_{i,j}^{(w)}$.

The algorithm to compute $s_i$ proceeds as follows. Start by computing $q_{i,0}^{(w)}$; using $q_{i,0}^{(w)}$ compute $q_{i,1}^{(w)}$; using $q_{i,0}^{(w)}, q_{i,1}^{(w)}$ compute $q_{i,2}^{(w)}$; and so on.

The computation of $q_{i,0}^{(w)}$ does not require any squaring or multiplication. For $1 \leq j \leq \tau_i$, the computation of $q_{i,j}^{(w)}$ requires $\mathfrak{r}_i$[S]+$j$[M]. So, the computation of all the quantities $q_{i,0}^{(w)}, q_{i,1}^{(w)}, \ldots, q_{i,\tau_i}^{(w)}$ requires $\tau_i \mathfrak{r}_i$[S]+$(\tau_i(\tau_i + 1)/2)$[M]. In addition, $\tau_i w$ squarings are required to compute the required powers of $\alpha_i$. The total number of operations required by $\mathsf{eval}(\alpha_i)$ is $(\tau_i w + \tau_i \mathfrak{r}_i)$[S]+$\tau_i(\tau_i + 1)/2$[M].

Algorithm $\mathsf{findSqRoot}$ makes a total of $k$ calls to $\mathsf{eval}$ on the $k$ inputs $\alpha_0, \ldots, \alpha_{k-1}$. The total number of operations in all of these calls is

$$\left(w \left(\sum_{i=0}^{k-1} \tau_i\right) + \mathfrak{r}\right) [\mathrm{S}] + \left(\sum_{i=0}^{k-1} \frac{\tau_i(\tau_i + 1)}{2}\right) [\mathrm{M}], \qquad (25)$$

where

$$\mathfrak{r} \;=\; \sum_{i=0}^{k-1} \tau_i \mathfrak{r}_i. \tag{26}$$

Since $\mathfrak{r}_i \leq w-1$ for $i = 0, \ldots, k-1$, a loose upper bound on $\mathfrak{r}$ is $(w-1)(\tau_0 + \ldots + \tau_{k-1}) < n$. For concrete cases, however, the value of $\mathfrak{r}$ can be significantly less. Consider for example, $n = 96$, $w = 6$ and $k = 2$. In this case, the value of $\mathfrak{r}$ is 7, whereas the upper bound is 96. For $n = 96$, $w = 4$ and $k = 4$, the value of $\mathfrak{r}$ is 5 whereas the upper bound is again 96.

**Remark:** Suppose $k = 1$ and $\ell_0 = n - 1$. Then $s_0 = 2q_0$. The strategy we have described above determines $q_0$. Instead, one may determine $s_0 = 2q_0$. Depending on the values of $n$ and $w$, this may be advantageous, since this may avoid the $\mathfrak{r}$ squarings required in the above strategy. Bernstein's method [4], determines $2q_0$. So, our method does not become Bernstein's method simply by setting $k = 1$ and $\ell_0 = n - 1$.

### 3.2 Computation of $g^t$

In findSqRoot, it is required to compute $g^{t_1}, \ldots, g^{t_{k-1}}$ and $g^{t_k/2}$, where $t_i$ is given by (5) to be $t_i = q_0 2^{\kappa_{i,0}} + q_1 2^{\kappa_{i,1}} + \cdots + q_{i-1} 2^{\kappa_{i,i-1}}$ and $q_0, \ldots, q_{i-1}$ are known.

In (6), the binary expansion of $q_j$ has been considered. For the table look-up based method, we consider the base $2^w$ expansion of $q_j$ as follows.

$$q_j \;=\; \mathfrak{q}_{j,0}^{(w)} + \mathfrak{q}_{j,1}^{(w)} 2^w + \cdots + \mathfrak{q}_{j,\tau_j}^{(w)} 2^{w\tau_j} \tag{27}$$

where $\mathfrak{q}_{j,0}^{(w)}, \mathfrak{q}_{j,1}^{(w)}, \ldots, \mathfrak{q}_{j,\tau_j-1}^{(w)} \in \{0, \ldots, 2^w - 1\}$ and $\mathfrak{q}_{j,\tau_j}^{(w)} \in \{0, \ldots, 2^{\ell_j - \tau_j w}\}$, i.e., $\mathfrak{q}_{j,0}^{(w)}, \mathfrak{q}_{j,1}^{(w)}, \ldots, \mathfrak{q}_{j,\tau_j-1}^{(w)}$ are $w$-bit integers whereas $\mathfrak{q}_{j,\tau_j}^{(w)}$ is an $(\ell_j - \tau_j w)$-bit integer. This is to be contrasted with the representation of $q_j$ used for computing eval given in (20), where the first coefficient is small and the rest of the coefficients are $w$-bit integers. If $w$ divides $\ell_j$, then the two representations coincide.

First consider the computation of $g^{t_i}$, for $i = 1, \ldots, k-1$. Using the expression for $t_i$ given by (5), this requires the computations of $g^{q_j 2^{\kappa_{i,j}}}$ for $j = 0, \ldots, i-1$. Using (27), we have

$$g^{q_j 2^{\kappa_{i,j}}} \;=\; g^{(\mathfrak{q}_{j,0}^{(w)} + \mathfrak{q}_{j,1}^{(w)} 2^w + \cdots + \mathfrak{q}_{j,\tau_j}^{(w)} 2^{w\tau_j}) 2^{\kappa_{i,j}}} \tag{28}$$

$$\;=\; \left( \left( g^{\mathfrak{q}_{j,0}^{(w)} 2^{w\zeta_{i,j}}} \right) \cdot \left( g^{\mathfrak{q}_{j,1}^{(w)} 2^{w(1+\zeta_{i,j})}} \right) \cdots \left( g^{\mathfrak{q}_{j,\tau_j}^{(w)} 2^{w(\tau_j+\zeta_{i,j})}} \right) \right)^{2^{\rho_{i,j}}}. \tag{29}$$

For the last step, we have used the relation $\kappa_{i,j} = w\zeta_{i,j} + \rho_{i,j}$ from (19). Note that for $j = 0, \ldots, i-1$, $q_j$ is known and so $\mathfrak{q}_{j,0}, \ldots, \mathfrak{q}_{j,\tau_j}$ are also known. Given the values of $\mathfrak{q}_{j,0}^{(w)}, \mathfrak{q}_{j,1}^{(w)}, \ldots, \mathfrak{q}_{j,\tau_j}^{(w)}$ each of the terms in the inner expression of (29) can be obtained from $\mathsf{Tab}_2$. So, the inner expression can be obtained using $\tau_j$ multiplications. Computing the whole expression in (29) requires an additional $\rho_{i,j}$ squarings.

For $i = 1, \ldots, k-1$, to compute $g^{t_i}$, it is required to obtain $g^{q_j 2^{\kappa_{i,j}}}$ for $j = 0, \ldots, i-1$ and then use $i-1$ multiplications to multiply these together. So, computing $g^{t_i}$ requires

$$\left( \textstyle\sum_{j=0}^{i-1} \rho_{i,j} \right) [\mathrm{S}] + \left( i - 1 + \textstyle\sum_{j=0}^{i-1} \tau_j \right) [\mathrm{M}].$$

The computation of $g^{t_k/2}$ is similar, except that $\kappa_{k,j} - 1$ is used in place of $\kappa_{i,j}$. Note that in (19), $\zeta_{k,j}$ and $\rho_{k,j}$, $j = 0, \ldots, k-1$ have been defined in a manner such that (29) holds with $i = k$.

The total number of operations to compute $g^{t_i}$ for $i = 1, \ldots, k-1$ and $g^{t_k/2}$ is the following.

$$\rho[\mathrm{S}] + \left( \sum_{i=0}^{k-1} (k-i)\tau_i + \frac{k(k-1)}{2} \right) [\mathrm{M}], \tag{30}$$

where

$$\rho \;=\; \sum_{i=1}^{k} \sum_{j=0}^{i-1} \rho_{i,j}. \tag{31}$$

Since $\rho_{i,j} \leq w - 1$, a loose upper bound on $\rho$ is $(w-1)k(k-1)$. For concrete cases, the value of $\rho$ can be significantly less. Consider for example, $n = 96$, $w = 6$ and $k = 2$. In this case, the value of $\rho$ is 6, whereas the upper bound is 10. Again, for $n = 96$, $w = 4$ and $k = 4$, the value of $\rho$ is 12, whereas the upper bound is 36.

### 3.3 Complexity

Apart from the computations of eval and the exponentiations to the power of $g$, the operations given by (12) are required. Combining these operation counts with the operation counts in the previous two sections, we have the following result.

**Theorem 5** *Let $w$ be a parameter. Using the table look-up method, the numbers of multiplications and squarings required by* findSqRoot *other than those considered in $\mathfrak{T}$ (for the computation of $v$) is as follows.*

$$\textit{Squarings:} \qquad \left( n - \ell_0 + w \left( \sum_{i=0}^{k-1} \tau_i \right) + \mathfrak{r} + \rho \right) \tag{32}$$

$$\textit{Multiplications:} \qquad \left( k + 2 + \left( \sum_{i=0}^{k-1} (k-i)\tau_i \right) + \sum_{i=0}^{k-1} \frac{\tau_i(\tau_i+1)}{2} + \frac{k(k-1)}{2} \right) [M] \tag{33}$$

$$\tag{34}$$

*where $\mathfrak{r}$ and $\rho$ are given by* (26) *and* (31) *respectively.*

*If $w$ divides $n$, then the pre-computed tables store $(2^w - 1)(n/w + 1)$ elements, while if $w$ does not divide $n$, then the pre-computed tables store $(2^w - 1)(\tau^\star + \lceil n/w \rceil)$ elements of $\mathbb{Z}_p$. Computing the tables require $(2^w - 1)n$ operations (i.e., squarings and multiplications).*

The following result determines the asymptotic complexity of the method.

**Theorem 6** *Given $w > 1$, it is possible to choose the parameters $k$ and $\ell_0, \ldots, \ell_{k-1}$ such that the total number of multiplications and squarings required by* findSqRoot *using table look-up is $\mathfrak{T} + O((n/w)^{3/2})$, where $\mathfrak{T}$ is the total number of multiplications and squarings required to compute $u^{(m-1)/2}$.*

**Proof:** Let $k = \lfloor \sqrt{(n-1)/w} \rfloor$ so that $k = O(\sqrt{n/w})$. Given $n$ and $k$, choose $\ell_0, \ldots, \ell_{k-1}$ as in (9). Then all of the quantities $\ell_0, \ldots, \ell_{k-1}$ are $O(\sqrt{nw})$ and so $\tau_0, \ldots, \tau_{k-1}$ are all $O(\sqrt{n/w})$. With these choices, the sum of the expressions given in (32) and (33) becomes $O((n/w)^{3/2})$ and so the corresponding time complexity of findSqRoot is $\mathfrak{T} + O((n/w)^{3/2})$. □

### 3.4 Concrete Comparison

First we compare the storage requirement of the new algorithm with that of Bernstein's algorithm [4]. Bernstein's method also requires pre-computed tuples of both the forms $(\nu, i, g^{-\nu 2^{iw}})$ and $(\nu, j, g^{-\nu 2^{n-jw}})$, $\nu \in \{1, \ldots, 2^w - 1\}$, $i = 0, \ldots, \lceil n/w \rceil - 1$ and $j = 2, \ldots, \lceil n/w \rceil - 1$. Additionally, Bernstein's method requires a table of pairs $(h^\nu, \nu)$, $\nu \in \{1, \ldots, 2^w - 1\}$ which corresponds to $\mathsf{Tab}_3$. So, the total storage required by Bernstein's method is $(2^w - 1)(2\lceil n/w \rceil - 1)$ tuples. If $w$ divides $n$, then the two tables storing tuples of the forms $(\nu, i, g^{\nu 2^{iw}})$ and $(\nu, j, g^{\nu 2^{n-jw}})$ become the same and so, the storage requirement becomes $(2^w - 1)(n/w + 1)$. In the example given in [4], $n = 96$ and $w$ is chosen to be either 6 or 8 so that $w$ divides $n$.

For the new algorithm, the total storage requirement is $(2^w - 1)(\tau^\star + \lceil n/w \rceil)$ tuples; if $w$ divides $n$, then $\mathsf{Tab}_1$ becomes a part of $\mathsf{Tab}_2$ and the total storage requirement is $(2^w - 1)(n/w + 1)$ tuples. So, if either $w$ divides $n$, or $k = 1$, then both Bernstein's method and the new method require the same storage. In general, since $\tau^\star \leq \lceil n/w \rceil - 1$, the new method will never require more storage than Bernstein's method and will require less storage whenever $\tau^\star < \lceil n/w \rceil - 1$.

Both Bernstein's algorithm and the new algorithm require $(2^w - 1)n$ operations for preparing the pre-computed tables.

Next we consider the comparison of the number of operations of the new algorithm and Bernstein's algorithm. Both Bernstein's algorithm and the new algorithm require the exponentiation $u^{(m-1)/2}$. So, the discussion on concrete comparison of Bernstein's algorithm with the new algorithm does not include the number of operations required for this exponentiation.

Bernstein's method requires $(n - w)$[S]$+\big(2 + \frac{1}{2} \cdot \lceil \frac{n}{w} \rceil \left(\lceil \frac{n}{w} \rceil + 1\right)\big)$[M] operations. The number of operations required by the new algorithm is given by Theorem 5. For concrete values of $n$, a comparison of the two algorithms is shown in Table 2. A cell in the table corresponds to a value of $n$ and a value of $w$. Each cell has two entries one above the other.

1. The upper entry in each cell corresponds to Bernstein's algorithm and has three components: the first component provides the numbers of squarings and multiplications, the second component is the total number of operations, and the third component provides a number $a$ such that $a(2^w - 1)$ elements of $\mathbb{Z}_p$ are to be stored.

2. The lower entry in each cell corresponds to the new algorithm and has four components. The first component is the value of $k$. Using this value of $k$ and $n$, the values of $\ell_0, \ldots, \ell_{k-1}$ have been chosen as in (9). The second component provides the numbers of squarings and multiplications, the third component provides the total number of operations, and the fourth component provides a number $a$ such that $a(2^w - 1)$ elements of $\mathbb{Z}_p$ are to be stored.

The entries for the new algorithm were generated using a Python program to compute the expressions given by (32) and (33). The value of $k$ was varied from 1 to $\sqrt{n}$, and among the various counts, the one for which the total number of operations is minimum has been reported in Table 2.

Based on the table, we have the following observations regarding the comparison between the new algorithm and Bernstein's algorithm.

1. For $w = 8$ and $n = 96$ or $n = 128$, Bernstein's method requires fewer squarings.

2. For $n = 96$ and $w = 6$, the new algorithm requires 146[S]+82[M] for a total of 228 operations, while Bernstein's algorithm requires 90[S]+138[M] which also leads to a total of 228 operations. However, since more squarings and fewer multiplications are required by the new algorithm, in practice it will be faster than Bernstein's algorithm.

Table 2: Comparison of Bernstein's table look-up method with the new table look-up method.

| | $w = 2$ | $w = 4$ |
|---|---|---|
| $n = 96$ | (94[S]+1178[M],1272,49)<br>(6,182[S]+338[M],520,49) | (92[S]+302[M],394,25)<br>(4,170[S]+122[M],292,25) |
| $n = 128$ | (126[S]+2082[M],2208,65)<br>(9,239[S]+514[M],753,65) | (124[S]+530[M],654,33)<br>(4,228[S]+194[M],422,33) |
| $n = 256$ | (254[S]+8252[M],8512,129)<br>(14,519[S]+1484[M],2003,129) | (252[S]+2082[M],2334,65)<br>(8,484[S]+514[M],998,65) |
| $n = 512$ | (510[S]+32898[M],33408,257)<br>(17, 991[S]+4098[M],5089,257) | (508[S]+8258[M],8766,129)<br>(8,972[S]+1538[M],2501,129) |
| $n = 1024$ | (1022[S]+131330[M],132352,513)<br>(19,2087[S]+11801[M],13888,513) | (1020[S]+32898[M],33918,257)<br>(16,1996[S]+4098[M],6094,257) |

| | $w = 6$ | $w = 8$ |
|---|---|---|
| $n = 96$ | (90[S]+138[M],228,17)<br>(2,146[S]+82[M],228,17) | (88[S]+80[M],168,13)<br>(1,100[S]+80[M],180,13) |
| $n = 128$ | (122[S]+255[M],377,42)<br>(3,235[S]+115[M],350,29) | (120[S]+138[M],258,17)<br>(1,136[S]+138[M],274,17) |
| $n = 256$ | (250[S]+948[M],1198,84)<br>(4,469[S]+332[M],801,53) | (248[S]+530[M],778,33)<br>(4,448[S]+194[M],642,33) |
| $n = 512$ | (506[S]+3743[M],4249,170)<br>(5,1057[S]+955[M],2012,103) | (504[S]+2082[M],2586,65)<br>(8,960[S]+514[M],1474,65) |
| $n = 1024$ | (1018[S]+14708[M],15726,340)<br>(16,2241[S]+2378[M],4619,188) | (1016[S]+8258[M],9274,129)<br>(8,1928[S]+1538[M],3466,129) |

3. In all other cases, the number of operations required by the new algorithm is less than that of Bernstein's algorithm. For a fixed value of $w$, the gap increases as the value of $n$ increases.

4. If $w$ divides $n$, then both the new algorithm and Bernstein's algorithm require the same storage. On the other hand, if $w$ does not divide $n$, the storage requirement of the new algorithm is less than that of Bernstein's algorithm.

Suppose $w$ is high enough compared to $n$ such that $k = 1$ is the best choice for the new algorithm. Setting $k = 1$ does not result in the new algorithm becoming the same as Bernstein's algorithm. We refer to the note at the end of Section 3.1. From Table 2, the cases arising from $w = 8$ and $n = 96$ or $n = 128$ have $k = 1$ as the best choice for the new algorithm. In both these cases, we note that Bernstein's algorithm performs better. It is not, however, true that when $k = 1$ for the new algorithm, Bernstein's algorithm will always perform better. For $n = 97$ and $w = 8$, Bernstein's algorithm requires 89[S]+93[M] for a total of 182 operations, while the new algorithm has $k = 1$ and requires 100[S]+80[M] for a total of 180 operations. Similar small improvements are observed for other values of $n$ and $w$.

**Remark:** For performing various kinds of asymptotic comparisons, Bernstein [4] sets $w \approx c \lg \lg p - d \lg \lg \lg p$, where $c$ and $d$ are constants with $c > 0$ and $d \geq 0$. Similar asymptotic comparisons can be carried out for the new algorithm. Since we have made a detailed comparison to Bernstein's algorithm, such comparisons are redundant and so we omit them.

## 4 Comparison to Müller's Algorithm

As mentioned in the introduction, for $p \equiv 3 \bmod 4$ or, $p \equiv 5 \bmod 8$, a single exponentiation in $\mathbb{Z}_p$ suffices to compute a square root. No other algorithm is faster. The comparison below is for primes congruent to 9 mod 16, i.e., $n \geq 3$.

For $p \equiv 9 \bmod 16$, i.e., $n = 3$, the algorithm by Kong et al. [9] requires 1[$\mathfrak{T}$]+2[S]+1[M] operations for half of the squares, while for the other half of the squares, it requires 2[$\mathfrak{T}$]+4[S]+7[M] operations, where $\mathfrak{T}$ is the number of operations required for raising to the power $(m-1)/2$. For a random square, the expected number of operations is 1.5[$\mathfrak{T}$]+3[S]+4[M]. The expected number of operations required by the Tonelli-Shanks algorithm for $n = 3$ is 6.75 operations plus 1[$\mathfrak{T}$] [11] and the expected number of operations required by the new algorithm is 1[$\mathfrak{T}$]+1.5[S]+4[M]. So, the expected number of operations required by the algorithm due to Kong et al. is more than the expected number of operations required by the Tonelli-Shanks algorithm and the new algorithm.

The TS algorithm and Bernstein's table look-up based method is one option for computing square roots for general $p$. As discussed in the introduction, for $n \geq 4$, the only competition to the TS/Bernstein algorithm is the Lucas sequence based algorithm due to Müller [12]. In the previous sections, we have already provided a detailed comparison of the number of operations required by the new algorithm to the TS/Bernstein algorithm. In this section, we compare to Müller's algorithm.

The algorithm proposed by Müller requires $-1 + \lg(p-1)$ squarings and the same number of multiplications to compute a square root. For the sake of simplicity, let us assume that this is $2 \lg p$ operations. Note that $\lg p \approx n + \lg m$. If $\lg m$ is very small in comparison to $\lg p$, then Müller's algorithm performs better than all other algorithms. Below, we indicate the range of $n$ for which the new algorithm requires fewer operations than Müller's algorithm.

Recall that the total number of operations (i.e., counting both multiplications and squarings) required for the exponentiation $u^{(m-1)/2}$ is about $(1+c)\lg m$ for some $c \in (0, 1/2]$. Using an appropriate choice of $k$ as in the proof of Theorem 3, the new algorithm requires about $n^{3/2}$ additional operations (i.e., counting both squarings and multiplications). So, the overall number of operations required by the new algorithm is about $(1+c)\lg m + n^{3/2}$. For $n^{3/2}/(1-c) - ((1+c)/(1-c))n \leq \lg p$, the number of operations required by the new algorithm is less than the number of operations required by Müller's algorithm[4]. If the table look-up based method is used, then the total number of operations required is about $(1+c)\lg m + (n/w)^{3/2}$, and so for $(n/w)^{3/2}/(1-c) - ((1+c)/(1-c))n \leq \lg p$ the number of operations required by the table look-up based method is less than the number of operations required by Müller's algorithm.

We provide a concrete example. As before, consider the prime $p = 2^{224} - 2^{96} + 1$ with $n = 96$ and $m = 2^{128} - 1$. For this prime, Müller's algorithm requires 223[S]+223[M]. (The algorithm also requires about 1.5 Legendre symbol evaluation, which we are ignoring for the purpose of this comparison). Now consider the new algorithm. The exponentiation $u^{(m-1)/2}$ requires 126[S]+10[M] (see [4]). Using the table look-up based implementation with $w = 4$ and $k = 4$, the number of additional operations required is 170[S]+122[M] for a total of 296[S]+132[M] with storage of 375 elements of $\mathbb{Z}_p$; with $w = 6$ and $k = 2$, the number of additional operations required is 146[S]+82[M] for a total of 272[S]+92[M] with storage of 1088 elements of $\mathbb{Z}_p$. Both of these options are faster than Müller's algorithm.

The comparison of the new algorithm with Müller's algorithm is based on the counts of the num-

[4]The condition $n^{3/2}/(1-c) - ((1+c)/(1-c))n \leq \lg p$ is equivalent to $m \geq 2^{(n^{3/2}-2n)/(1-c)}$; by Dirichlet's theorem, for every $n$, the set $\{1 + i2^n : i \geq 2^{(n^{3/2}-2n)/(1-c)}\}$ contains infinitely many primes, and so, for every $n$, there are infinitely many primes for which the new algorithm improves upon Müller's algorithm.

bers of multiplications and squarings required by the two algorithm. While this is indicative, careful implementations are required to determine the practical efficiency gains. We leave such work for the future[5].

## Acknowledgement

Thanks to Dan Brown and Ying Tong for pointing out typos and to the reviewers for their kind comments.

## References

[1] Leonard M. Adleman, Kenneth L. Manders, and Gary L. Miller. On taking roots in finite fields. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 175–178. IEEE Computer Society, 1977.

[2] A. O. L. Atkin. Probabilistic primality testing, summary by F. Morain. Technical Report 1779, INRIA, 1992.

[3] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory, Volume 1, Efficient Algorithms*. MIT Press, 1996.

[4] Daniel J. Bernstein. Faster square roots in annoying finite fields. `https://cr.yp.to/papers.html#sqroot`, 2001.

[5] Daniel J. Bernstein. Pippenger's exponentiation algorithm. `https://cr.yp.to/papers.html#pippenger`, 2002.

[6] Zhengjun Cao, Qian Sha, and Xiao Fan. Adleman-Manders-Miller root extraction method revisited. In Chuankun Wu, Moti Yung, and Dongdai Lin, editors, *Information Security and Cryptology - 7th International Conference, Inscrypt 2011, Beijing, China, November 30 - December 3, 2011. Revised Selected Papers*, volume 7537 of *Lecture Notes in Computer Science*, pages 77–85. Springer, 2011.

[7] M. Cipolla. Un metodo per la risolutione della congruenza di secondo grado. *Rendiconto dell'Accademia Scienze Fisiche e Matematiche, Napoli, Series 3*, IX:154–163, 1903.

[8] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag Berlin Heidelberg, 1993.

[9] Fanyu Kong, Zhun Cai, Jia Yu, and Daxing Li. Improved generalized Atkin algorithm for computing square roots in finite fields. *Information Processing Letters*, 98(1):1–5, 2006.

[10] Derrick H. Lehmer. Computer technology applied to the theory of numbers. In W. Leveque, editor, *MAA Studies in number theory, 6*, pages 117–151. Prentice-Hall, Englewood Cliffs, New Jersey, 1969.

[11] Scott Lindhurst. An analysis of Shanks algorithm for computing square roots in finite fields. In *CRM Proceedings and Lecture Notes*, volume 19, pages 231–242. 1999.

[5] We note that an independent prototype implementation of the new algorithm is available at `https://github.com/zcash/pasta/blob/master/squareroottab.sage`.

[12] Siguna Müller. On the computation of square roots in finite fields. *Des. Codes Cryptogr.*, 31(3):301–312, 2004.

[13] Armand Stefan Rotaru and Sorin Iftene. A complete generalization of Atkin's square root algorithm. *Fundam. Informaticae*, 125(1):71–94, 2013.

[14] D. Shanks. Five number-theoretic algorithms. In R. S. D. Thomas and Hugh C. Williams, editors, *Proceedings of the second Manitoba conference on numerical mathematics*, Congressus Numerantium 7, Utilitas Mathematica, pages 51–70, 1973.

[15] Alberto Tonelli. Bemerkung über die auflösung quadratischer congruenzen. *Göttinger Nachrichten*, pages 344–346, 1891.

# A Relation to the Tonelli-Shanks and the Adleman-Manders-Miller Algorithms

As before, suppose the square root of $u$ is to be computed, and let $v = u^{(m-1)/2}$ and $x = u \cdot v^2 = u^m$. By Lemma 1, there is a non-negative integer $t$ such that $x \cdot g^t = 1$ and then $u \cdot v \cdot g^{t/2}$ is a square root of $u$. The goal of both the TS and the AMM algorithm is essentially to compute $g^{t/2}$. We briefly explain how the two algorithms perform this computation.

The Tonelli-Shanks algorithm computes $g^{t/2}$ as follows. Since, $x^{2^{n-1}} = 1$, by repeatedly squaring $x$, an integer $i_1 \in \{0, \ldots, n-2\}$ is obtained such that $x^{2^{i_1}} = -1$. Since $g^{2^{n-1}} = -1$, we have $x^{2^{i_1}} \cdot g^{2^{n-1}} = 1$. Let $c_1 = g^{2^{n-2-i_1}}$ and $y_1 = x \cdot c_1^2$. We have $y_1^{2^{i_1}} = 1$. If $y_1 = 1$, then stop; otherwise, by repeatedly squaring $y_1$, an integer $i_2 \in \{0, \ldots, i_1 - 1\}$ is obtained such that $y_1^{2^{i_2}} = -1$. Again $y_1^{2^{i_2}} \cdot g^{2^{n-1}} = 1$. Let $c_2 = c_1 \cdot g^{2^{n-2-i_2}}$ and $y_2 = x \cdot c_2^2 = y_1 \cdot g^{2^{n-1-i_2}}$ so that $y_2^{2^{i_2}} = 1$. If $y_2 = 1$, then stop; otherwise, by repeatedly squaring $y_2$ obtain $i_3 \in \{0, \ldots, i_2 - 1\}$ such that $y_2^{2^{i_3}} = -1$. The algorithm proceeds by defining $c_3, c_4, \ldots$ and correspondingly $y_3, y_4, \ldots$. Since $n - 1 > i_1 > i_2 > i_3 > \cdots$, the process stops at some step $r$ such that $y_r = 1$. At this point, we have $1 = y_r = x \cdot c_r^2$. The element $c_r$ is the required quantity $g^{t/2}$. An algorithmic description of the Tonelli-Shanks algorithm can be found in [8, 3].

The Adleman-Manders-Miller algorithm computes $g^{t/2}$ as follows. The powers $x^2, x^{2^2}, \ldots, x^{2^{n-2}}$ and the powers $g^2, g^{2^2}, \ldots, g^{2^{n-2}}$ are computed and stored. Let $w_0 = x^{2^{n-1}} = 1$. A square root of $w_0$ is $y_0 = x^{2^{n-2}} \in \{1, -1\}$. Let $a_0 \in \{0, 1\}$ be such that $y_0 = (-1)^{a_0}$. Let, $w_1 = y_0 \cdot g^{a_0 2^{n-1}} = 1$. Since, $w_1 = y_0 \cdot g^{a_0 2^{n-1}} = x^{2^{n-2}} \cdot g^{a_0 2^{n-1}}$ a square root of $w_1$ is $y_1 = x^{2^{n-3}} \cdot g^{a_0 2^{n-2}} \in \{1, -1\}$ which can be obtained from the computed powers of $x$ and $g$. Let $a_1 \in \{0, 1\}$ be such that $y_1 = (-1)^{a_1}$. Let, $w_2 = y_1 \cdot g^{a_1 2^{n-1}} = 1$. Since $w_2 = y_1 \cdot g^{a_1 2^{n-1}} = x^{2^{n-3}} \cdot g^{a_0 2^{n-2}} \cdot g^{a_1 2^{n-1}}$, a square root of $w_2$ is $y_2 = x^{2^{n-4}} \cdot g^{a_0 2^{n-3}} \cdot g^{a_1 2^{n-2}} \in \{1, -1\}$ which can be obtained from the computed powers of $x$ and $g$. The procedure continues by obtaining $w_3, y_3$, $w_4, y_4$ and so on until it terminates after $n - 1$ steps yielding $w_{n-1} = x \cdot g^{a_0 2 + a_1 2^2 + \cdots + a_{n-2} 2^{n-1}} = 1$. Then $t = a_0 2 + a_1 2^2 + \cdots + a_{n-2} 2^{n-1}$ and $g^{t/2} = (g)^{a_0} \cdot \left(g^2\right)^{a_1} \cdot \left(g^{2^2}\right)^{a_2} \cdots \left(g^{2^{n-2}}\right)^{a_{n-2}}$ which can be computed using the previously computed powers of $g$. A description of this idea can be found in [6].

Algorithm findSqRoot uses the parameter $k$. Depending on the value of $k$, the new algorithm becomes essentially the TS algorithm or essentially the AMM algorithm. We explain this below.

Suppose $k$ is chosen to be 1. Then $\ell_0 = n - 1$ and $x_0 = x$. So, $\alpha_0 = x_0$. The loop in Steps 7 to 9 has only one iteration, in which eval($\alpha_0$) computes and returns the value $s_0$. In Step 10, $t_1$ is set to be equal to $t_0 + s_0 = s_0$ (since $t_0 = 0$) and $\gamma$ is computed to be $g^{t_1/2} = g^{s_0/2}$. The computed square root

of $u$ is $u \cdot v \cdot \gamma = u \cdot v \cdot g^{s_0/2}$. The TS algorithm also returns $u \cdot v \cdot g^{s_0/2}$. The description of the TS algorithm in [8] is a little different, where $s_0$ is not explicitly computed. Exercise 25 of [8], on the other hand, suggests that $s_0$ can indeed be explicitly computed. So, for $k = 1$, the new algorithm provides a different formulation of the TS algorithm.

Suppose $k$ is chosen to be $n - 1$. Then $\ell_0 = \cdots = \ell_{k-1} = 1$. The loop in Steps 7 to 9 iterates for $n - 1$ steps. The calls to $\mathsf{eval}$ in these $n - 1$ steps are $\mathsf{eval}(\alpha_0), \mathsf{eval}(\alpha_1), \ldots, \mathsf{eval}(\alpha_{k-1})$ which returns $s_0, \ldots, s_{k-1}$ respectively. From Lemma 3, we have $1 = \alpha_i^{2^{\ell_i}} = \alpha_i^2$, for $i = 0, \ldots, k - 1$. So, each $\alpha_i$ is either 1 or $-1$. Further, Lemma 3 also shows that $\alpha_i \cdot g^{s_i} = 1$. So, if $\alpha_i = 1$, then $s_i = 0$, and if $\alpha_i = -1$, then $s_i = 2^{n-1}$. In other words, if we write $\alpha_i = (-1)^{a_i}$, then $s_i = a_i 2^{n-1}$. The value of $t_i$ is computed as $(t_{i-1} + s_{i-1})/2 = (t_{i-1} + a_i 2^{n-1})/2$. So, the value of $t_k = t_{n-1}$ computed in Step 10 is equal to $a_0 2 + a_1 2^2 + \cdots + a_{n-2} 2^{n-1}$. Next $g^{t_k/2}$ is computed. Recalling the description of the AMM algorithm provided above, we see that choosing $k = n - 1$ leads to exactly the AMM algorithm.

So, at the two ends, i.e., for $k = 1$ and $k = n - 1$, the new algorithm is essentially the TS and the AMM algorithms respectively. For intermediate choices of $k$, the new algorithm provides improved complexity.