Optimal 2-3 Chains for Scalar Multiplication

Cristobal Leiva and Nicolas Thériault*

Departamento de Matemática y ciencia de la Computación,
Universidad de Santiago de Chile,
Santiago, Chile
{nicolas.theriault,cristobal.leiva}@usach.cl

Abstract. Using double-base chains to represent integers, in particular chains with bases 2 and 3, can be beneficial to the efficiency of scalar multiplication. However, finding an optimal 2-3 chain as long been thought to be more expensive than the scalar multiplication itself, complicating the use of 2-3 chains in practical applications where the scalar is used only a few time (as in the Diffie-Hellman key exchange).
In the last few years, important progress has been made in obtaining the shortest possible double-base chain for a varying integer n. In 2008, Doche and Habsieger used a binary-tree based approach to get a (relatively close) approximation of the minimal chain. In 2015, Capuñay and Thériault presented the first deterministic polynomial-time algorithm to compute the minimal chain for a scalar, but the complexity of $O((\log n)^{3+\epsilon})$ is too high for use with a varying scalars. More recently, Bernstein, Chuengsatiansup, and Lange used a graph-based approach to obtain an algorithm with running time $O((\log n)^{2.5+\epsilon})$.
In this work, we adapt the algorithm of Capuñay and Thériault to obtain minimal chains in $O((\log n)^2 \log\log n)$ bit operations and $O((\log n)^2)$ bits of memory. This allows us to obtain minimal chains for 256-bits integers in the 0.338 millisecond range, making it useful to reduce scalar multiplication costs randomly-selected scalars.
We also show how to extend the result to other types of double-base and triple-base chains (although the complexity for triple-base chains is cubic instead of quadratic). In the case of environments with restricted memory, our algorithm can be adapted to compute the minimal chain in $O((\log n)^2(\log\log n)^2)$ bit operations with only $O(\log n(\log\log n)^2)$ bits of memory.

Keywords: Integer representations, double-base chains, scalar multiplication.

1 Introduction

Scalar multiplication is an integral part of group-based cryptosystems, and ever since the introduction of elliptic curves for cryptographic applications [12, 11], there has been constant efforts to improve the efficiency of scalar multiplication,

* This research was supported by FONDECYT grant 1151326 (Chile).

mainly by improving the group operations or using a representation of the secret scalar n that reduces the number of these group operations.

In 1998, Dimitrov et al. [8] proposed the use of double-base chains to reduce the cost of scalar multiplications in certain groups. Several families of elliptic curves are known to be interesting for double-base chains in base 2-3 as they have a very favorable ratio between the costs of group doublings and triplings [7, 6, 2, 3]. An obvious problem with the implementation of such systems is the need to find a chain with low Hamming weight, ideally one with the minimal possible Hamming weight.

An exhaustive search is clearly out of the question, so alternative approaches need to be found. Although a greedy algorithm easily produces double-base chains, they are far from optimal. The first step to obtaining optimal chains was taken by Doche and Habsieger [9], producing much better chains than the greedy approach, although the chains produced are still slightly sub-optimal in general.

In recent years, two approaches have been proposed to find optimal chains in polynomial time, first by Capuñay and Thériault, with complexity $O((\log n)^{3+\epsilon})$, and then by Bernstein, Chuengsatiansup, and Lange [4], decreasing the complexity to $O((\log n)^{2.5+\epsilon})$.

This paper aims at reducing the cost of obtaining optimal double base chains, lowering it to $O((\log n)^2 \log\log n)$ bit operations. Reducing to this complexity should be considered essential to the application of double-base chains in cases where the scalar changes at every use or after a fixed number of uses, as in the Diffie-Hellman key exchange.

The paper is organized as follows: a background on double-base chains and the algorithms to compute them is given in Section 2. In Section 3 we show how to improve the algorithm of Capuñay and Thériault to obtain quadratic complexity. We then provide generalizations to other double-base systems in Section 4 (and triple-base in Appendix A) and in Section 5 we show how to adapt the algorithm to restricted-memory environments at the cost of an extra $\log\log n$ factor in complexity. We give experimental results for the fastest form of the algorithm in Section 6, showing how effective it can be even at various integer sizes. We conclude in Section 7.

2 Background

2.1 Double-Base Chains

Definition 1. *Given p and q, two distinct prime numbers, a double-base number system (DBNS), is a representation scheme in which every positive integer n is represented as the sum or difference of numbers of the form $p^a q^b$, i.e.*

$$n = \sum_{i=1}^{m} s_i p^{a_i} q^{b_i}, \text{ with } s_i \in \{-1, 1\}, \text{ and } a_i, b_i \geq 0. \tag{1}$$

The Hamming weight of a DBNS representation is the number m of terms in (1).

The main interest of DBNS comes from the possibility of decreasing the Hamming weight of the representation of the scalar n. In [8], Dimitrov *et al.* showed that for any integer n, it is possible to use a greedy algorithm to obtain a DBNS expansion of n having at most $O\left(\frac{\log n}{\log\log n}\right)$ terms.

However, this representation is usually not suited for scalar multiplications as it minimizes the number of group additions but does not worry about the number of multiplications by p and q required to reach these additions. In effect, the DBNS representations of n with minimized Hamming weight will typically be more costly then the corresponding single-base representations. To avoid this problem, Dimitrov *et al.* [7] proposed the use of double-base chains.

Definition 2. *A double-base chain (DBC) for n (or simply called a p-q chain for n) is an expansion of the form*

$$n = \sum_{i=1}^{m} s_i p^{a_i} q^{b_i}, \text{ with } s_i \in \{-1, 1\}, \tag{2}$$

such that $a_1 \geq a_2 \geq \cdots \geq a_m \geq 0$ and $b_1 \geq b_2 \geq \cdots \geq b_m \geq 0$.

The monotone form of the sequence of exponents makes it more easily applicable to scalar multiplication by minimizing the number of multiplications by p and q while still reducing the Hamming weight.

Note that most DBNS representations cannot be written as chains since the degrees in a DBNS representation can vary independently (for example $59 = 2^5 + 3^3$), so chains are more restrictive representations.

The simplest case (in terms of group operations) comes from 2-3 chains. There are several example of algebraic groups where the ratio between the costs of doublings and tripling are close enough to the theoretical proportion of $\ln 2 / \ln 3$ for an ideal interchange between the two bases. Experimental results in [5] show that the Hamming weight for 2-3-chains is lower than other integer recodings with the same digit sets, but still linear in $\log n$, coming close to $0.19 \log_2 n$.

A number of technique to obtain double-base chains of low Hamming weight have been developed in the last few years. In the following subsections, we describe the algorithm of Capuñay and Thériault (which will be the basis of our result) and give a brief overview of alternative approaches.

2.2 Algorithm of Capuñay and Thériault

We now describe the algorithm of Capuñay and Thériault, simplifying some of the notation to the minimal required to follow the current work. The proofs of the different results can be found in [5].

Definition 3. *Given a positive integer n, we denote by $n_{i,j}$ the (unique) integer in $\{0, 1, \ldots, (2^i 3^j - 1)\}$ such that $n_{i,j} \equiv n \bmod 2^i 3^j$, and by $\overline{n}_{i,j}$ the (unique) integer in $\{-1, -2, \ldots, -2^i 3^j\}$ such that $\overline{n}_{i,j} \equiv n \bmod 2^i 3^j$.*

Note that this definition differs slightly from that of [5] (where $\overline{n}_{i,j}$ is the (unique) integer in $\{0, -1, \ldots, -(2^i 3^j - 1)\}$ such that $\overline{n}_{i,j} \equiv n \bmod 2^i 3^j$). We will discuss this change in Remark 1, however, this new definition clearly implies that $\overline{n}_{i,j} = n_{i,j} - 2^i 3^j$, so the discussion can naturally be restricted to the positive values $(n_{i,j})$.

Definition 4. *We denote by $\mathcal{C}_{i,j}$ a minimal 2-3 chain for $n_{i,j}$ in which all terms are strict divisors of $2^i 3^j$ (that is to say, any term in the chain is of the form $\pm 2^a 3^b$ with at least $a < i$ or $b < j$), and by $\overline{\mathcal{C}}_{i,j}$ a minimal 2-3 chain for $\overline{n}_{i,j}$ in which all terms are strict divisors of $2^i 3^j$. If no such chain is possible, we write $\mathcal{C}_{i,j} = \emptyset$, or $\overline{\mathcal{C}}_{i,j} = \emptyset$.*

Note that there may be several choices of optimal chains for $\mathcal{C}_{i,j}$ and/or $\overline{\mathcal{C}}_{i,j}$ if more than one chain has the same (minimal) Hamming weight. However, all of them are equivalent and interchangeable in terms of the desired solution, so only one needs be taken into account. We also write $\emptyset \pm 2^i 3^j = \emptyset$ since it is not possible to extend a non-existing chain.

To ensure the first steps of recursive arguments follow the general pattern, we also write $\mathcal{C}_{-1,j} = \overline{\mathcal{C}}_{-1,j} = \mathcal{C}_{i,-1} = \overline{\mathcal{C}}_{i,-1} = \emptyset$.

The inductive process of the algorithm of Capuñay and Thériault relies on the observation that subchains of minimal chains must also be minimal (for the corresponding $n_{i,j}$ or $\overline{n}_{i,j}$. It also uses the recursive relations between the values $n_{i,j}$ and $\overline{n}_{i,j}$. By definition of $n_{i,j}$ and $\overline{n}_{i,j}$, it is clear that if $i > 0$ then

$$n_{i,j} \in \{n_{i-1,j}, n_{i-1,j} + 2^{i-1}3^j\} \quad , \quad \overline{n}_{i,j} \in \{\overline{n}_{i-1,j}, \overline{n}_{i-1,j} - 2^{i-1}3^j\}$$

and if $j > 0$ then

$$\begin{aligned} n_{i,j} &\in \{n_{i,j-1}, n_{i,j-1} + 2^i 3^{j-1}, n_{i,j-1} + 2 \cdot 2^i 3^{j-1}\} \\ \overline{n}_{i,j} &\in \{\overline{n}_{i,j-1}, \overline{n}_{i,j-1} - 2^i 3^{j-1}, \overline{n}_{i,j-1} - 2 \cdot 2^i 3^{j-1}\} \end{aligned}$$

The algorithm looks at the evolution of the values of $n_{i,j}$ and $\overline{n}_{i,j}$. However, under the definition of double-base chains used, terms of the form $\pm 2 \cdot 2^i 3^{j-1}$ are not allowed in a 2-3 chain (unless we write them as $\pm 2^{i+1} 3^{j-1}$), so they incompatible with our definition of $\mathcal{C}_{i,j}$ and $\overline{\mathcal{C}}_{i,j}$.

Given a chain for $n_{i,j}$ (where all terms are decreasing divisors $2^i 3^j$, all smaller than $2^i 3^j$ in absolute value), we can extract a chain for one of $n_{i-1,j}$, $\overline{n}_{i-1,j}$, $n_{i,j-1}$ or $\overline{n}_{i,j-1}$, and this gives us inductive relationships.

Lemma 1. *[Lemma 3 of [5]] $\mathcal{C}_{i,j} \neq \emptyset$ if and only if one (or more) of the following cases occurs:*

1. $\mathcal{C}_{i,j} = \mathcal{C}_{i-1,j}$ *(only if $n_{i,j} = n_{i-1,j}$);*
2. $\mathcal{C}_{i,j} = 2^{i-1}3^j + \mathcal{C}_{i-1,j}$ *(only if $n_{i,j} = 2^{i-1}3^j + n_{i-1,j}$);*
3. $\mathcal{C}_{i,j} = 2^{i-1}3^j + \overline{\mathcal{C}}_{i-1,j}$ *(only if $n_{i,j} = 2^{i-1}3^j + \overline{n}_{i-1,j}$);*
4. $\mathcal{C}_{i,j} = \mathcal{C}_{i,j-1}$ *(only if $n_{i,j} = n_{i,j-1}$);*
5. $\mathcal{C}_{i,j} = 2^i 3^{j-1} + \mathcal{C}_{i,j-1}$ *(only if $n_{i,j} = 2^i 3^{j-1} + n_{i,j-1}$);*
6. $\mathcal{C}_{i,j} = 2^i 3^{j-1} + \overline{\mathcal{C}}_{i,j-1}$ *(only if $n_{i,j} = 2^i 3^{j-1} + \overline{n}_{i,j-1}$);*

Similar cases occur for negative chains (after interchanging the signs).

Proof. See [5], Lemma 3.

Lemma 1 gives relations between the chains $C_{i,j}$ in terms of decreasing values of i or j (or both). However, to construct the chains it is more convenient to build relations on increasing values of i and j, and hence (in general) increasing values of $n_{i,j}$ (since optimal chains are more easily obtained for smaller values of n).

However, increasing the values of the pair (i, j) has a disadvantage: whereas subchains of an optimal chain are also optimal, the same does not necessarily hold when extending a subchain. As a result, the extended subchains will be candidates for the optimal chains $\mathcal{C}_{i,j}$ and $\overline{\mathcal{C}}_{i,j}$, leading to the following definition:

Definition 5. *A double-base chain C is called a* (possible) source *for $\mathcal{C}_{i,j}$ (resp. $\overline{\mathcal{C}}_{i,j}$) if C sums up to $n_{i,j}$ (resp. $\overline{n}_{i,j}$), and one of the following holds:*

- *If the largest term is of the form $\pm 2^{i-1}3^j$, then the subchain obtained from C by removing this term is optimal for $n_{i-1,j}$ or $\overline{n}_{i-1,j}$;*
- *If the largest term is of the form $\pm 2^i 3^{j-1}$, then the subchain obtained from C by removing this term is optimal for $n_{i,j-1}$ or $\overline{n}_{i,j-1}$;*
- *If the largest term is neither of the form $\pm 2^{i-1}3^j$ or $\pm 2^i 3^{j-1}$, then C is optimal for either $n_{i-1,j}$ or $n_{i,j-1}$ (resp. $\overline{n}_{i-1,j}$ or $\overline{n}_{i,j-1}$).*

The set of possible sources for $\mathcal{C}_{i,j}$ is denoted $\mathcal{S}_{i,j}$ (resp. $\overline{\mathcal{S}}_{i,j}$ for $\overline{\mathcal{C}}_{i,j}$).

Although not all sources are optimal chains for $n_{i,j}$ (resp. $\overline{n}_{i,j}$), as soon as there are some sources at least one of them must be optimal, since the subchains of an optimal chain are optimal and therefore can produce a source of $\mathcal{C}_{i,j}$ (resp. $\overline{\mathcal{C}}_{i,j}$).

Remark 1. Our use of Definition 3 rather than that of [5] has the following motivation. For $n_{i,j} \neq 0$, both definitions coincide, so the only difference comes for $n_{i,j} = 0$.

- From both definitions, $\mathcal{C}_{i,j} = 0$ if and only if $n_{i,j} = 0$.
- From the definition of $\overline{n}_{i,j}$ in [5], $\overline{\mathcal{C}}_{i,j} = 0$ if and only if $\overline{n}_{i,j} = 0 = n_{i,j}$.
- $\mathcal{C}_{i,j} = 0$ and $\overline{\mathcal{C}}_{i,j} = 0$ are completely interchangeable in the construction of larger chains, and we can safely assume that priority is given to the positive chain.
- From Definition 3, if $n_{i,j} = 0$, then $\overline{n}_{i,j} = -2^i 3^j$ so $\overline{\mathcal{C}}_{i,j} = \emptyset$: consecutive terms in a chain grow (in absolute value) by a factor of at least 2 and the largest one is $\leq 2^{i-1}3^j$ (in absolute value), so the sum of the chain cannot sum up to $-2^i 3^j$.

The change in definition will therefore have no impact on the terms of the chain produced by the algorithm.

With our definition of $n_{i,j}$ and $\overline{n}_{i,j}$, the corollary that allows the construction of optimal chains from [5] becomes:

Table 1. Possible sources of $\mathcal{C}_{i,j}$ and $\overline{\mathcal{C}}_{i,j}$ when multiplying by 2.

$n_{i,j}$	Possible $\mathcal{C}_{i,j}$	Possible $\overline{\mathcal{C}}_{i,j}$
$n_{i-1,j}$	$\mathcal{C}_{i-1,j}$, $2^{i-1}3^j + \overline{\mathcal{C}}_{i-1,j}$	$-2^{i-1}3^j + \overline{\mathcal{C}}_{i-1,j}$
$n_{i-1,j} + 2^{i-1}3^j$	$2^{i-1}3^j + \mathcal{C}_{i-1,j}$	$\overline{\mathcal{C}}_{i-1,j}$, $-2^{i-1}3^j + \mathcal{C}_{i-1,j}$

Table 2. Possible sources of $\mathcal{C}_{i,j}$ and $\overline{\mathcal{C}}_{i,j}$ when multiplying by 3.

$n_{i,j}$	Possible $\mathcal{C}_{i,j}$	Possible $\overline{\mathcal{C}}_{i,j}$
$n_{i,j-1}$	$\mathcal{C}_{i,j-1}$, $2^i3^{j-1} + \overline{\mathcal{C}}_{i,j-1}$	$\emptyset$
$n_{i,j-1} + 2^i3^{j-1}$	$2^i3^{j-1} + \mathcal{C}_{i,j-1}$	$-2^i3^{j-1} + \overline{\mathcal{C}}_{i,j-1}$
$n_{i,j-1} + 2 \cdot 2^i3^{j-1}$	$\emptyset$	$\overline{\mathcal{C}}_{i,j-1}$, $-2^i3^{j-1} + \mathcal{C}_{i,j-1}$

Corollary 1. *[Corollary 1 of [5]] Given chains $\mathcal{C}_{i-1,j}$, $\overline{\mathcal{C}}_{i-1,j}$, $\mathcal{C}_{i,j-1}$ and $\overline{\mathcal{C}}_{i,j-1}$, then the possible sources for $\mathcal{C}_{i,j}$ and $\overline{\mathcal{C}}_{i,j}$ can be found in Tables 1 and 2.*

Proof. See [5], Corollary 1, adjusting to Definition 3.

In order to simplify the final step of our algorithm, we introduce one more notation:

Definition 6. *For every integer $0 \leq j \leq \lceil \log_3 n \rceil$, we set the (optimal) chains C_j and $\overline{C}_j$ as $C_j = \mathcal{C}_{i,j}$ and $\overline{C}_j = 2^i3^j + \overline{\mathcal{C}}_{i,j}$, where i is the smallest non-negative integer such that $2^i3^j > n$.*

Algorithm 1 compute optimal subchains $\mathcal{C}_{i,j}$ (for $n_{i,j}$) and $\overline{\mathcal{C}}_{i,j}$ (for $\overline{n}_{i,j}$) for increasing values of i until $n \leq 2^i3^j < 2n$ (after which $\mathcal{C}_{i,j}$ remains fixed) and $\overline{C}_j$ is replaced by $2^i3^j + \overline{\mathcal{C}}_{i,j}$.

Theorem 1. *[Therorem 1 of [5]] The set S defined as*

$$S = \{C_j \mid 0 \leq j \leq \lceil \log_3(n+1) \rceil\} \cup \{\overline{C}_j \mid 0 \leq j \leq \lceil \log_3(n+1) \rceil\}$$

obtained from Algorithm 1 contains a minimal 2-3 chain for n.

Proof. See [5], Theorem 1.

Theorem 2. *[Theorem 2 of [5]] Let n be a positive integer, then Algorithm 1 returns a minimal 2-3 chain in $O((\log n)^4)$ bit operations ($O((\log n)^{3+\epsilon})$ if fast arithmetic is used), and requires $O((\log n)^3)$ bits of memory.*

Proof. See [5], Theorem 2.

Algorithm 1: Algorithm to compute a minimal 2-3 chain.

Input: Integer $n > 0$.
Output: Minimal 2-3 chain $\mathcal{C}$ for n.

$C_j \leftarrow \emptyset$, $\overline{C}_j \leftarrow \emptyset$ for every j
for $i \leftarrow 0$ ***to*** $\lceil \log_2(n+1) \rceil$ **do**
  $m \leftarrow \lceil \log_3((n+1)/2^i) \rceil$, $i_m \leftarrow \lceil \log_2((n+1)/3^m) \rceil$
  if $i_m < i$ **then**
    $m \leftarrow m - 1$
  for $j \leftarrow 0$ ***to*** m **do**
    $n_{i,j} \leftarrow n \bmod 2^i 3^j$
    $\mathcal{P}_{i,j} \leftarrow \emptyset$, $\overline{\mathcal{P}}_{i,j} \leftarrow \emptyset$
    if $i > 0$ **then**
      $n_{i-1,j} \leftarrow n \bmod 2^{i-1} 3^j$
      if $n_{i,j} = n_{i-1,j}$ **then**
        Include C_{i-1} and $2^{i-1}3^j + \overline{C}_{i-1}$ in $\mathcal{P}_{i,j}$
        Include $-2^{i-1}3^j + \overline{C}_{i-1}$ in $\overline{\mathcal{P}}_{i,j}$
      else
        Include $2^{i-1}3^j + C_{i-1}$ in $\mathcal{P}_{i,j}$
        Include $\overline{C}_{i-1}$ and $-2^{i-1}3^j + C_{i-1}$ in $\overline{\mathcal{P}}_{i,j}$
    if $j > 0$ **then**
      $n_{i,j-1} \leftarrow n \bmod 2^i 3^{j-1}$
      if $n_{i,j} = n_{i,j-1}$ **then**
        Include C_i and $2^{i-1}3^j + \overline{C}_{i-1}$ in $\mathcal{P}_{i,j}$
        No change to $\overline{\mathcal{P}}_{i,j}$
      else if $n_{i,j} = n_{i,j-1} + 2^i 3^{j-1}$ **then**
        Include $2^i 3^{j-1} + C_i$ in $\mathcal{P}_{i,j}$
        Include $-2^{i-1}3^j + \overline{C}_i$ in $\overline{\mathcal{P}}_{i,j}$
      else
        No change to $\mathcal{P}_{i,j}$
        Include $\overline{C}_i$ and $-2^{i-1}3^j + C_i$ in $\overline{\mathcal{P}}_{i,j}$
    $C_i \leftarrow$ shortest chain in $\mathcal{P}_{i,j}$ and update its length
    if $i = j = 0$ **then**
      $C_0 \leftarrow 0$ of length 0
    $\overline{C}_i \leftarrow$ shortest chain in $\overline{\mathcal{P}}_{i,j}$ and update its length
  if $i = i_m$ **then**
    $\overline{C}_m \leftarrow 2^i 3^m + \overline{C}_m$.
$\mathcal{C} \leftarrow$ shortest chain among the C_j and $\overline{C}_j$
return $\mathcal{C}$

2.3 Other Approaches

Over the years, two other approaches have been proposed to compute double-base chains. The first one, by Doche and Habsieger [9], uses a binary tree to compute the shortest double-base chain it can find (under the restriction doublings and/or triplings are given priority over group additions, producing slightly sub-optimal chains). As pointed out in [4], this algorithm can be made polynomial time if it is taken into account that the values of the nodes correspond to the $n_{i,j}$ and $\overline{n}_{i,j}$ in Algorithm 1, and hence there are polynomialy many of them. The tree should therefore keep track of all the values encountered at previous level (discarding repeated ones) to limit its growth. Although the complexity of this approach can be controlled, the sub-optimality of the result does not appear to be avoidable with a binary tree-based approach.

A more recent algorithm by Bernstein, Chuengsatiansup, and Lange [4] uses a graph-based approach to search for the optimal chain. This algorithm produces optimal chains just as Algorithm 1, but its complexity is $O((\log n)^{2.5+\epsilon})$, making it more efficient. Nonetheless, it will still grow asymptotically faster than scalar multiplication with fast arithmetic techniques ($O((\log n)^{2+\epsilon})$).

3 Reducing the Complexity

In this section, we describe our improvements to the algorithm of Capuñay and Thériault to obtain quadratic complexity. For the description and analysis, we consider the whole set of steps (optimal subchains $\mathcal{C}_{i,j}$ and $\overline{\mathcal{C}}_{i,j}$ for all valid pairs (i,j)), referring to *horizontal* steps those corresponding to doublings (increase in the index i) and *vertical* steps those corresponding to triplings (increase in the index j). This horizontal-vertical look will simplify the generalization to other basis in the next section.

Even if the algorithm of Capuñay and Thériault does not keep information about previous subchains (in fact, it overwrites these subchains), we will sometime consider that they are all available for the sake of discussion while developing the new algorithm, and then clarify what minimal information needs to be kept.

3.1 Reduced Memory by Retracing the Steps

Since our goal is to obtain an algorithm with at most quadratic complexity, a first problem encountered with Algorithm 1 comes from obtaining $O(\log n)$ final chains, each of $O(\log n)$ terms (from the growth of the Hamming weight) of size $O(\log n)$, for a total memory cost of $O((\log n)^3)$.

Obviously this must be reduced before we can hope to come close to quadratic time complexity. Keeping the terms of the chains solely in terms of their exponents reduces the memory to $O((\log n)^2 \log n)$, but we can do better.

Rather than keep complete minimal chains at every step (removing or overwriting the previous incomplete chain), we can keep track of all the movements

in the two-dimensional array corresponding to the algorithm steps. The information required at each step is:

- Whether the chain comes from a horizontal or vertical step (1 bit);
- Whether the previous chain was a positive or negative chain (1 bit);
- Whether the current chain has the same terms or one more term than the previous chain (1 bit);
- (If the number of terms increased), the sign of the latest term (1 bit).

Note that if the current chain has one more term than the previous chain, then the sign of the term is clear from the current and previous positions (knowing whether they are positive or negative chains). We could therefore record all the information for each step with 3 bits instead of 4.

The information is allocated as a block for each case of the step (for example, "15" for a chain due to a vertical move from a negative chain, adding a negative term), so the cost (in time) is the same for 3 or 4 bits of information. We preferred to use 4 bits to simplify the backtracking process and because of a more natural fit into the architecture.

This new *movements array* takes $\approx 4 \log_2 n \log_3 n$ bits (in a rectangular array, $\approx 2 \log_2 n \log_3 n$ if it can be stored in a triangular array corresponding to the maximal exponents). The associated Hamming weight array would be of size $O((\log n)^2 \log\log n)$ due to the bound on the exponents, however the algorithm only needs to keep track of the Hamming weights for two of the inner loops at a time since the optimal chains for a pair (i, j) only depend on the chains for $(i-1, j)$ and $(i, j-1)$, giving us size $O((\log n) \log\log n)$ for the Hamming weights.

Once we know which chain has the lowest Hamming weight, it can be written by retracing our steps (backtracking through the new *movements array*), which takes time $O(\log n \log\log n)$. That is to say: Step 34 compares $O(\log n)$ values of size $\leq \log_2 n$, and then Step 35 retraces the the selected chain to write it out.

As an added bonus, the computational complexity, over the whole algorithm, spent on keeping track of the shortest chains (Steps 28, 31, and 33 of Algorithm 1) also decrease to $O((\log n)^2 \log\log n)$.

3.2 Order of the steps

In [5], the double "for" loop (Steps 2 and 6) is done such that all the subchains for a given i (power of 2) are computed before increasing the value of i. For all intents and purposes, this ordering has little effect on the algorithm, except on the number of subchains for a given index. By setting the first loop in i rather than j, then the number of (positive and negative) subchains to store is $\approx 2 \log_3 n$ instead of $\approx 2 \log_2 n$, which gives a small reducing in memory.

For our improvements to work efficiently, we first invert this order, computing all the subchains for a given j (power of 3) before increasing the value of j. The reason for this choice will become clear in the next subsections.

In term of notation, the algorithm will then be written in terms of C_i and $\overline{C}_i$ instead C_j and $\overline{C}_j$, with the equivalent definition:

Definition 7. *For every integer $j \geq 0$ such that $2^i 3^j < 4n$, we denote by C_i and $\overline{C}_i$ the minimal positive and negative subchains for the current $n_{i,j}$ and $\overline{n}_{i,j}$. If $2^i 3^j \geq 4n$, then C_i remains unchanged from previous values of j.*

We could bound j such that $2^i 3^j \leq 2n$ as a parallel to Definition 6, however the bound $2^i 3^j < 4n$ (which leads to the final j satisfying $2n \leq 2^i 3^j < 4n$) has the advantage that the negative chain $\overline{\mathcal{C}}_{i-1,j}$ for $n_{i-1,j} = n$ produces a chain $2^{i-1}3^j + \overline{\mathcal{C}}_{i-1,j}$ for n. As a result, the negative subchains $\overline{C}_i$ need not be considered in the final step of the algorithm.

3.3 Efficient Computation of the Possible Sources

To reduce the computational complexity, we first look at how the different cases for the doubling steps (going from $(i-1, j)$ to (i, j), Steps 10 to 16 of Algorithm 1) can be distinguished more efficiently.

Identifying the case of horizontal steps (increasing the index i) consists in determining if $n_{i,j} = n_{i-1,j}$ or $n_{i-1,j}+2^{i-1}3^j$. If all the values of $n_{i,j}$ or computed separately, each cost $O((\log n)^{1+\epsilon})$ for a total cost of $O((\log n)^{3+\epsilon})$ over the whole algorithm.

Since $n_{i-1,j} \equiv n \bmod 2^{i-1}3^j$ and $n_{i,j} \equiv n \bmod 2^i 3^j$, then

$$\begin{aligned} \frac{n_{i,j} - n_{i-1,j}}{2^{i-1}3^j} &= \frac{\big(n - \big(n \bmod 2^{i-1}3^j\big)\big) \bmod 2^i 3^j}{2^{i-1}3^j} \\ &= \big(n \operatorname{div} 2^{i-1}3^j\big) \bmod 2 \\ &= (i-1)\text{-th bit of } (n \operatorname{div} 3^j). \end{aligned} \tag{3}$$

For a given j, we can then extract all the horizontal steps (sources of minimal chains coming from doublings) from the binary representation of $n \operatorname{div} 3^j$. Since the algorithm already does a recursion in j (from 0 to m), we compute the binary expansions of $n \operatorname{div} 3^j$ as

$$n \operatorname{div} 3^j = (n \operatorname{div} 3^{j-1}) \operatorname{div} 3,$$

which requires a total of m divisions by 3. Furthermore, division by 3 can be implemented in linear time since we are dealing with a small, fixed denominator. We can therefore compute all the binary expansions of the form $n \operatorname{div} 3^j$ in time $O((\log n)^2)$ bit operations (without requiring fast arithmetic techniques).

Also note that the largest value of i to consider for the current j is obtained directly from the binary representation of $n \operatorname{div} 3^j$: it is simply the number of bits in this binary representation.

Steps 10 to 16, identifying the possible horizontal sources for $\mathcal{C}_{i,j}$ and $\overline{\mathcal{C}}_{i,j}$, can then be completed in $O(\log \log n)$ time ($O(1)$ time except for the Hamming weight counter which is $\leq \log_2 n$). Over the whole algorithm, the cost associated to Steps 10 to 16 then decreases to $O((\log n)^2 \log \log n)$.

3.4 Using Only the Binary Representations

To identify the cases associated to tripling steps (changes in j, Steps 18 to 27 of Algorithm 1), we could apply the same approach as in the previous subsection, working in base 3 instead of base 2, again obtaining quadratic time. However, the following lemma shows how to obtain the same result directly from the binary expansions of n div 3^j.

Lemma 2. *Let a_i and b_i be the i-th bits of $(n \text{ div } 3^{j-1})$ and $(n \text{ div } 3^j)$ respectively:*

- *If $a_i + b_i \equiv 1 \bmod 2$, then $n_{i,j} = n_{i,j-1} + 2^i 3^{j-1}$;*
- *If $a_i + b_i \equiv 0 \bmod 2$, then $n_{i,j} = n_{i,j-1}$ or $n_{i,j-1} + 2 \cdot 2^i 3^{j-1}$:*
  - *If $a_i + a_{i+1} + b_{i+1} \equiv 0 \bmod 2$, then $n_{i,j} = n_{i,j-1}$;*
  - *If $a_i + a_{i+1} + b_{i+1} \equiv 1 \bmod 2$, then $n_{i,j} = n_{i,j-1} + 2 \cdot 2^i 3^{j-1}$.*

Proof. Let $m = n \text{ div } (2^i 3^{j-1})$. By a similar argument as in the previous section, $(n_{i,j} - n_{i,j-1})/2^i 3^{j-1}$ is equal to $m \bmod 3$, so the binary expansion of $a = n \text{ div } 3^{j-1}$ contains the essential information about the tripling case. Since $n \text{ div } (2^i 3^j) = m \text{ div } 3$, we can also use the binary expansion of $b = n \text{ div } 3^j$ to help identify the current tripling case without doing divisions by 3 for position i. Then $a_i \equiv m \bmod 2$, $b_i \equiv (m \text{ div } 3) \bmod 2$, $a_{i+1} \equiv (m \text{ div } 2) \bmod 2$, and $b_{i+1} \equiv (m \text{ div } 6) \bmod 2$, which naturally leads us to consider $m \bmod 12$.

$m \bmod 12$	0	1	2	3	4	5	6	7	8	9	10	11
$m \bmod 3$	0	1	2	0	1	2	0	1	2	0	1	2
a_i	0	1	0	1	0	1	0	1	0	1	0	1
b_i	0	0	0	1	1	1	0	0	0	1	1	1
a_{i+1}	0	0	1	1	0	0	1	1	0	0	1	1
b_{i+1}	0	0	0	0	0	0	1	1	1	1	1	1

We observe that $a_i \neq b_i$ if and only if $m \equiv 1 \bmod 3$, which allows us to identify the first case. To distinguish the other two cases (where $a_i = b_i$), we note that $a_{i+1} = b_{i+1}$ if $a_i = 0$ and $m \equiv 0 \bmod 3$ or $a_i = 1$ and $m \equiv 1 \bmod 3$. Similarly, $a_{i+1} \neq b_{i+1}$ if $a_i = 1$ and $m \equiv 0 \bmod 3$ or $a_i = 0$ and $m \equiv 1 \bmod 3$. The statement of the lemma then follows directly. □

An important effect of this lemma is that we can determine all the cases for doubling (horizontal) steps and tripling (vertical) steps for a given value of j knowing only the states (Hamming weight of the subchains) of the previous value of j and the binary expansions of $n \text{ div } 3^{j-1}$ and $n \text{ div } 3^j$. As a result, the memory that would be required to run the algorithm without retracing the chains (i.e. finding only the Hamming weight of the chain) is $O(\log n \log\log n)$ bits (since the Hamming weight are all bounded by $\log_2 n$).

3.5 Algorithm

Combining the improvements in the previous subsections, we obtain Algorithm 2. To simplify the pseudocode (so all values of i and j are treated uniformly), we write a_{-1} for the -1-th bit (NULL), and bits of the NULL value (also NULL), under the assumption that NULL bits in a conditional statement implies the algorithm skips the whole statement. The discussion leading to Algorithm 2 leads directly to its complexity:

Theorem 3. *Let n be a positive integer, then Algorithm 2 returns a minimal 2-3 chain in $O((\log n)^2 \log\log n)$ bit operations, and requires $O((\log n)^2)$ bits of memory.*

Proof. The complexity is straightforward: the double loop in i and j runs through $O((\log n)^2)$ steps. As explained, each step requires $O(\log\log n)$ bit operations for Hamming weight updates – all other operations (bit extraction and comparisons for case selections and updates of the *movements array*) being $O(1)$.

The correctness of the algorithm is a direct consequence of Theorem 1, the changes having to do with the order in which the loops are performed (which does not affect the operations at each step), how the different cases are distinguished (given by Equation 3 and Lemma 2), and representation of the chains (through the *movements array*). □

Note that the actual implementation benefits from the following ideas (although these do not change the asymptotic complexity):

- Instead of building the sets $\mathcal{P}_{i,j}$ and $\overline{\mathcal{P}}_{i,j}$, each new source is compared the previous best (initially to $\emptyset$) and replaces it if it is shorter.
- Rather than looking at chains going to the position (i, j), we look at chains produced by that position (sources coming from $\mathcal{C}_{i,j}$ and $\overline{\mathcal{C}}_{i,j}$):
  - This allows us to avoid dealing with special cases when positions with $i = -1$ and $j = -1$ would be required.
  - Doubling steps from (i, j) to $(i+1, j)$ are performed first (thus finalizing the work for position $(i + 1, j)$), after which tripling steps from (i, j) to $(i, j + 1)$ (giving the initial sources for position $(i, j + 1)$).
  - Only the bits a_i, b_i, a_{i+1} and b_{i+1} are used at position (i, j), reducing the number of bit extractions.
- Step 28 is in fact performed on-the-go at the end of Step 4, once the last C_i for the current j has been obtained. This also simplifies determining the starting position in the *movements array* at Step 29, by recording the pair (i, j) for this minimal Hamming weight.
- While performing Step 4, we check if all the current Hamming weights for the C_i and $\overline{C}_i$ associated to j are bigger then the current candidate for the optimal chain. If this occurs, then the loop in j is terminated as none of the remaining chains can have lower Hamming weight.

The final change has little impact on the asymptotic cost of the algorithm, but in practice it usually reduces the running time by 3 to 9% (the savings are more important for smaller values of n).

Algorithm 2: Algorithm to compute a minimal 2-3 chain.

Input: Integer $n > 0$.
Output: Minimal 2-3 chain $\mathcal{C}$ for n.

$C_i \leftarrow \emptyset$, $\overline{C}_i \leftarrow \emptyset$ for every i, all with length NULL
$C_0 \leftarrow 0$ with length 0, $a \leftarrow n$, $b \leftarrow NULL$
  $[j = 0]$
while $a > 0$ **do**
  for $i \leftarrow 0$ ***to*** size(a) **do**
    $a_{i-1} \leftarrow (i-1)$-th bit of a, $a_i \leftarrow i$-th bit of a, $a_{i+1} \leftarrow (i+1)$-th bit of a
    $b_i \leftarrow i$-th bit of b, $b_{i+1} \leftarrow (i+1)$-th bit of b
    $\mathcal{P}_i \leftarrow \emptyset$, $\overline{\mathcal{P}}_i \leftarrow \emptyset$ $[\mathcal{P}_i = \mathcal{P}_{i,j}, \overline{\mathcal{P}}_i = \overline{\mathcal{P}}_{i,j}]$
    if $a_{i-1} = 1$ **then**
      Include C_{i-1} and $2^{i-1}3^j + \overline{C}_{i-1}$ in $\mathcal{P}_i$ $[n_{i,j} = n_{i-1,j}]$
      Include $-2^{i-1}3^j + \overline{C}_{i-1}$ in $\overline{\mathcal{P}}_i$
    else
      Include $2^{i-1}3^j + C_{i-1}$ in $\mathcal{P}_i$ $[n_{i,j} = n_{i-1,j} + 2^{i-1}3^j]$
      Include $\overline{C}_{i-1}$ and $-2^{i-1}3^j + C_{i-1}$ in $\overline{\mathcal{P}}_i$
    if $a_i + b_i \equiv 1 \bmod 2$ **then**
      Include $2^i 3^{j-1} + C_i$ in $\mathcal{P}_i$ $[n_{i,j} = n_{i,j-1} + 2^i 3^{j-1}]$
      Include $-2^i 3^{j-1} + \overline{C}_i$ in $\overline{\mathcal{P}}_i$
    else
      if $a_i + a_{i+1} + b_{i+1} \equiv 0 \bmod 2$ **then**
        Include C_i and $2^i 3^{j-1} + \overline{C}_{i-1}$ in $\mathcal{P}_i$ $[n_{i,j} = n_{i,j-1}]$
        No change to $\overline{\mathcal{P}}_i$
      else
        No change to $\mathcal{P}_i$ $[n_{i,j} = n_{i,j-1} + 2 \cdot 2^i 3^{j-1}]$
        Include $\overline{C}_i$ and $-2^i 3^{j-1} + C_i$ in $\overline{\mathcal{P}}_i$
    $C_i \leftarrow$ shortest chain in $\mathcal{P}_i$ and update its length
    $\overline{C}_i \leftarrow$ shortest chain in $\overline{\mathcal{P}}_i$ and update its length
    Update the *movements array* as in Section 3.1
  $b \leftarrow a$
    $[j = j + 1]$
  $a \leftarrow a$ div 3
$C \leftarrow$ shortest chain in $\{C_i\}$
$\mathcal{C} \leftarrow$ retrace the steps for chain C using the *movements array*
return $\mathcal{C}$

Our C++ implementation of Algorithm 2 can be found at the following site: https://github.com/leivaburto/23chains/blob/master/23.cpp

4 Other Double-Base and Triple-Base Systems

It is relatively easy to adapt Algorithm 2 to other double bases. To do so, we first observe that Table 1 can be used for the steps in the construction of a $2-q$ chain (with $q > 3$) simply by replacing powers of 3 by powers of q. Similarly, Table 2 can be used for the steps in the construction of a $3-q$ chain (with $q > 3$) simply by replacing powers of 2 by powers of q (and changing the order of indices).

The equivalent to Corollary 1 for base q is the following lemma. Note that Tables 1 and 2 can be seen as collapsed cases of Table 3 (where some of the cases overlap if $q \leq 3$).

Lemma 3. *Given double-base chains $\mathcal{C}_{i-1,j}$, $\overline{\mathcal{C}}_{i-1,j}$, $\mathcal{C}_{i,j-1}$ and $\overline{\mathcal{C}}_{i,j-1}$ in bases p and q where the index j correspond to base q, then the possible sources for $\mathcal{C}_{i,j}$ and $\overline{\mathcal{C}}_{i,j}$ can be found in Table 3.*

Table 3. Possible sources of $\mathcal{C}_{i,j}$ and $\overline{\mathcal{C}}_{i,j}$ when multiplying by $q > 3$.

$n_{i,j}$	Possible $\mathcal{C}_{i,j}$	Possible $\overline{\mathcal{C}}_{i,j}$
$n_{i,j-1}$	$\mathcal{C}_{i,j-1}$, $p^i q^{j-1} + \overline{\mathcal{C}}_{i,j-1}$	$\emptyset$
$n_{i,j-1} + p^i q^{j-1}$	$p^i q^{j-1} + \mathcal{C}_{i,j-1}$	$\emptyset$
$n_{i,j-1} + (q-1) \cdot p^i q^{j-1}$	$\emptyset$	$\overline{\mathcal{C}}_{i,j-1}$, $-p^i q^{j-1} + \mathcal{C}_{i,j-1}$
$n_{i,j-1} + (q-2) \cdot p^i q^{j-1}$	$\emptyset$	$-p^i q^{j-1} + \overline{\mathcal{C}}_{i,j-1}$
other cases	$\emptyset$	$\emptyset$

Proof. Similar to the arguments for Corollary 1. □

We also observe that our these techniques can be applied to double-base chains that consider coefficient sets other than ± 1 for example $\{\pm 1, \pm 2, \ldots, \pm k\}$ (these chains have been studied in [10] and [4]). To do so, it would be sufficient to adjust Tables 1, 2, and 3 to include the new possible sources, and increase the number of layers for each coordinates (i, j) (each layer corresponding to a distinct nonzero coefficient) to consider chains for $n_{i,j}^{(\ell)} \equiv n \bmod p^i q^j$ in the interval $[(\ell-1)p^i q^j, \ell p^i q^j[$ (with $n_{i,j}^{(1)} = n_{i,j}$) and $\overline{n}_{i,j}^{(\ell)} \equiv n \bmod p^i q^j$ in the interval $]-\ell p^i q^j, -(\ell-1)p^i q^j]$ (with $\overline{n}_{i,j}^{(1)} = \overline{n}_{i,j}$). The increased number of layers is required to account for the increased number of final corrections to n, i.e. $n = n_{i,j}^{(\ell)} - (\ell-1)p^i q^j$ or $n = \overline{n}_{i,j}^{(\ell)} + \ell p^i q^j$.

In the following discussion, we only consider the detailed adaptation for the case $q = 5$ with the coefficient set ± 1.

4.1 2-5 Chains

When working in base 2-5, Algorithm 2 can be used almost as-is, replacing base 3 by base 5 and the application of Table 2 in Steps 14 to 23 by an application of Table 3 with $q = 5$. This means divisions by 3 in Step 28 are replaced with divisions by 5 (which still have cost linear in the size of a).

However, using the bits of a and b to extract the vertical cases in Steps 14 to 23 requires a complete re-write of Lemma 2:

Lemma 4. *Let a_i and b_i be the i-th bits of $(n \operatorname{div} 5^{j-1})$ and $(n \operatorname{div} 5^j)$ respectively:*

- *If $a_i + b_i \equiv 0 \bmod 2$, then $n_{i,j} = n_{i,j-1} k \cdot 2^i 5^{j-1}$ with $k \in \{0, 2, 4\}$:*
  - *If $a_{i+1} + b_{i+1} \equiv 1 \bmod 2$, then $n_{i,j} = n_{i,j-1} + 2 \cdot 2^i 5^{j-1}$;*
  - *If $a_{i+1} + b_{i+1} \equiv 0 \bmod 2$, then $n_{i,j} = n_{i,j-1}$ or $n_{i,j-1} + 4 \cdot 2^i 5^{j-1}$:*
    - *If $a_i + a_{i+2} + b_{i+2} \equiv 1 \bmod 2$, then $n_{i,j} = n_{i,j-1}$;*
    - *If $a_i + a_{i+2} + b_{i+2} \equiv 0 \bmod 2$, then $n_{i,j} = n_{i,j-1} + 4 \cdot 2^i 5^{j-1}$;*
- *If $a_i + b_i \equiv 1 \bmod 2$, then $n_{i,j} = n_{i,j-1} + 2^i 5^{j-1}$ or $n_{i,j-1} + 3 \cdot 2^i 5^{j-1}$:*
  - *If $a_i + a_{i+1} + b_{i+1} \equiv 1 \bmod 2$, then $n_{i,j} = n_{i,j-1} + 2^i 5^{j-1}$;*
  - *If $a_i + a_{i+1} + b_{i+1} \equiv 0 \bmod 2$, then $n_{i,j-1} = n_{i,j-1} + 3 \cdot 2^i 5^{j-1}$.*

Proof. Similar to the proof of Lemma 2, working modulo 40. □

Since all changes are $O(1)$ proportional to the work done in Algorithm 2, the cost of the resulting algorithm will still be quadratic in $\log n$ (with the same $\log \log n$ term due to keeping track of the Hamming weights).

4.2 3-5 Chains

To work in base 3-5, we use base-3 representations for n and $n \operatorname{div} 5^j$, whose trits (trinary digits) give the cases for the horizontal (tripling) steps in Table 2. To determine the cases in Table 3 with $q = 5$ (for the vertical/quintupling steps), we again re-work Lemma 2:

Lemma 5. *Let a_i and b_i be the i-th trits of $(n \operatorname{div} 5^{j-1})$ and $(n \operatorname{div} 5^j)$ respectively:*

- *If $a_i + b_i \equiv 2 \bmod 3$, then $n_{i,j} = n_{i,j-1} + 2 \cdot 3^i 5^{j-1}$;*
- *If $a_i + b_i \equiv 0 \bmod 3$, then $n_{i,j} = n_{i,j-1}$ or $n_{i,j-1} + 4 \cdot 3^i 5^{j-1}$:*
  - *If $(a_i + 1)^2 + a_{i+1} + b_{i+1} \equiv 1 \bmod 3$, then $n_{i,j} = n_{i,j-1}$;*
  - *If $(a_i + 1)^2 + a_{i+1} + b_{i+1} \equiv 2 \bmod 3$, then $n_{i,j} = n_{i,j-1} + 4 \cdot 3^i 5^{j-1}$;*
- *If $a_i + b_i \equiv 1 \bmod 3$, then $n_{i,j} = n_{i,j-1} + 3^i 5^{j-1}$ or $n_{i,j-1} + 3 \cdot 3^i 5^{j-1}$:*
  - *If $a_i^2 + a_{i+1} + b_{i+1} \equiv 1 \bmod 3$, then $n_{i,j} = n_{i,j-1} + 3^i 5^{j-1}$;*
  - *If $a_i^2 + a_{i+1} + b_{i+1} \equiv 0 \bmod 3$, then $n_{i,j} = n_{i,j-1} + 3 \cdot 3^i 5^{j-1}$;*

Proof. Similar to the proof of Lemma 2, working modulo 45. □

Once again, we obtain an algorithm of $O((\log n)^2 \log \log n)$ bit operations and $O((\log n)^3)$ bits of memory.

5 Implementation with Limited Memory

The memory requirements of Algorithm 2 may still be too high for practical application in certain cases: For n of common cryptographic sizes in embedded devices, the $O((\log n)^2)$ bits of memory may already impractical; and even for high-end computers, pushing beyond the 2^{16} bits range becomes highly problematic. Considering that almost all of the operations performed in the algorithm are at the bit-level (with the exception of keeping track of the Hamming weights), it would be reasonable to expect hardware implementations to perform even better than in software, but only once the memory has been reduced further.

In fact, this problem can be resolved rather easily, at the cost of a slight increase in complexity. As we stated in Section 3.4, the memory required for the algorithm to compute the minimal Hamming weight without retracing the actual chain is $O(\log n \log\log n)$, which is the amount of memory required to run Steps 4 to 28 for one value of j. Also, given the complete information for a given j, it would be possible to restart the algorithm at that point in order to finish the search. Furthermore, to retrace one step of a subchain from a given position (i, j) it is sufficient to have the running information for the current and previous value of j (in fact, that information would be enough to retrace all steps which go back at most one value of j).

These observation lead to a straightforward divide-and-conquer approach in which the length of the loop in j is divided in 2, and the information for the middle step is stored in memory. Repeating this inductively (on the second half of the interval when going forward, on the first half when backtracking), we are able to retrace the whole chain using the information for $O(\log\log n)$ values of j at any time, for a total of $O(\log n(\log\log n)^2)$ bits of memory. For a given j, this process performs the computations at most $O(\log\log n)$ times, so the complexity increases by a factor of $O(\log\log n)$ (to a total of $O((\log n)^2(\log\log n)^2)$).

Note that in some cases where the memory requirements are less limited, we could perform a one-off division of the interval into $O(\sqrt{\log n})$ sub-intervals and retrace the chain in at most twice the time of Algorithm 2, but with $O((\log n)^{1.5}\log\log n)$ memory.

6 Experimental Results

The $O((\log n)^2 \log\log n)$ complexity of the double-base chain search is certainly comparable to the best possible complexity for scalar multiplication, which is asymptotically at least $O((\log n)^2 \log\log n)$ assuming FFT multiplication is applicable on the underlying ring structures. Taking into account that for practical cryptographic sizes the asymptotic results for fast arithmetic for scalar multiplication are overly optimistic, whereas our complexity does not depend on fast arithmetic, one can reasonably expect the search algorithm to perform very well. It should also perform much better than alternative approach since its complexity grows much more slowly.

To determine the actual efficiency of the algorithm, we experimented it for different scalars of varying sizes. Table 4 gives the running time for integers of

sizes going up to 512 bits. We recall that for double-base chains to be competitive, the combined cost of obtaining the optimal chain and the scalar multiplication using that chain should be lower than the cost of a scalar multiplication using a single base representation (typically a NAF, taking into account the coefficient sets considered). The experiments were done on a 3.7 GHz Intel Xeon E5.

Since 2-3-chains appear to have Hamming density close to 0.19 instead of the 0.333 density of the NAF representation (see [5] for experimental results), using optimal double-base chains reduces the number of group additions by roughly 43%. For the use of double-base chains to be successful, the search for the optimal chain should take less than $0.143 \log_2 n$ group doublings. A complete cost comparison would depend on the specific group implementation, and is therefore beyond the scope of this paper. Instead, we report on direct computational costs.

The timings obtained in our experiments indicate that using double-base chains could indeed reduce costs for all but the fastest scalar multiplication implementations, and certainly for groups where 2-3 chains would be of interest (the fastest scalar multiplication implementations being in groups where doublings are particularly cheap).

Table 4. Average time (microseconds) to compute optimal 2-3-chains for n of k bits.

k	64	128	192	256	320	384	448	512
μsec	29.4	95.8	196.9	338.4	509.3	728.5	962.4	1245.0

To better illustrate the quadratic growth of the complexity, we performed experiments for integers of 2^ℓ bits, with ℓ going up to 15. The results can be seen in Table 5.

Table 5. Average time (milliseconds) to compute optimal 2-3-chains for n of k bits.

k	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
$m$sec	0.338	1.245	5.134	18.485	74.185	292.204	1167.301	4714.592

7 Conclusion

We presented a $O((\log n)^2 \log\log n)$ algorithm to compute optimal 2-3 chains that is significantly faster than scalar multiplication, so it can be used for on-the-go computation of the optimal chain as part as the overall scalar multiplication. This algorithm requires $O((\log n)^2)$ bits of memory, but the algorithm can be adapted to compute the optimal chain in time $O((\log n)^2(\log\log n)^2)$ with $O(\log n(\log\log n)^2)$ bits of memory.

We also extended the result to other double-base systems, especially bases 2-5 and 3-5, both with the same $O((\log n)^2 \log\log n)$ complexity. These results also extend to the search of optimal triple base chains for base 2-3-5 in time $O((\log n)^3 \log\log n)$ and memory $O((\log n)^3$.

Side-channel security (especially against SPA attacks) can a concern for the security of scalar multiplication when using double-base chains. In some groups,

uniform operations (using atomic blocks) without dummy operations have been constructed (see for example [1]). The design of such group operations is beyond the scope of this paper, and in a sense this paper could be seen as a pre-requisite do such work: in the past, improving the efficiency of obtaining optimal double-base chains has been a bottleneck to increase the interest in developing better (SPA-secure) triplings.

Acknowledgements

The authors would like to thanks the anonymous referees for their useful comments and suggestions.

References

1. R. Abarzúa and N. Thériault. *Complete Atomic Blocks for Elliptic Curves in Jacobian Coordinates over Prime Fields.* in: Progress in Cryptology – LATINCRYPT 2012, LNCS **7533**, pp. 37–55, Springer-Verlag, 2012.
2. D. J. Bernstein, P. Birkner, T. Lange, and C. Peters. *Optimizing Double-Base Elliptic-Curve Single-Scalar Multiplication.* in: Progress in Cryptology – INDOCRYPT 2007, LNCS **4859**, pp. 167–182, Springer-Verlag, 2007.
3. D. J. Bernstein, C. Chuengsatiansup, D. Kohel, and T. Lange. *Twisted Hessian curves.* in: Progress in Cryptology – LATINCRYPT 2015, LNCS **9230**, pp. 269–294, Springer-Verlag, 2015.
4. D. J. Bernstein, C. Chuengsatiansup, and T. Lange. *Double-base scalar multiplication revisited.* IACR eprint archive **2017/037**, 2017.
5. A. Capuñay and N. Thériault. *Computing Optimal 2-3 Chains for Pairings.* in: Progress in Cryptology – LATINCRYPT 2015, LNCS **9230**, pp. 225–244, Springer-Verlag, 2015.
6. M. Ciet, M. Joye, K. Lauter, and P. L. Montgomery. *Trading Inversions for Multiplications in Elliptic Curve Cryptography.* Designs, Codes and Cryptography, **39** (2), pp. 189–206, 2006.
7. V. Dimitrov, L. Imbert, and P. K. Mishra. *Efficient and Secure Elliptic Curve Point Multiplication Using Double-Base Chain.* in: Advances in Cryptology – ASIACRYPT 2005, LNCS **3788**, pp. 59–78, Springer-Verlag, 2005.
8. V. S. Dimitrov, G. A. Jullien, and W. C. Miller. *An algorithm for modular exponentiation.* Inform. Process. Lett., **66** (3), pp. 155–159, 1998.
9. C. Doche and L. Habsieger. *A Tree-Based Approach for Computing Double-Base Chains.* in: Information Security and Privacy – ACISP 2008, LNCS **5107**, pp. 433–446, Springer-Verlag, 2008.
10. C. Doche and L. Imbert. *Extended Double-Base Number System with Applications to Elliptic Curve Cryptography.* in: Progress in Cryptology – INDOCRYPT 2006, LNCS **4329**, pp. 335–348, Springer-Verlag, 2006.
11. N. Koblitz. *Elliptic Curve Cryptosystems.* Mathematics of Computation, **48**, pp. 203–209, 1987.
12. V. S. Miller. *Use of Elliptic Curves in Cryptography.* in: Advances in Cryptology – CRYPTO 1985, LNCS **218**, pp. 417-426, Springer-Verlag, 1986.

A Triple-base chains

The algorithms to obtain optimal 2-3-chains and 2-5-chains in Sections 3 and 4 can be combined to obtain a polynomial time algorithm to compute optimal tripple-base (2-3-5) chains for n, which is described in Algorithm 3.

Since we are now working in three dimension, each plane corresponding a coordinate k (the power of 5 in $2^i 3^j 5^k$) must have access to the subchains for $k-1$, so the array C_i is replaced by a double array $C_{i,j}$.

Theorem 4. *Let n be a positive integer, then Algorithm 3 returns a minimal 2-3-5 chain in $O((\log n)^3 \log\log n)$ bit operations, and requires $O((\log n)^3)$ bits of memory.*

Proof. Similar to the proof of Theorem 3.

The ideas of Section 5 can also be applied to Algorithm 3, reducing its memory requirements to $O((\log n)^2(\log\log n)^2)$ bits, at the expense of increasing its complexity to $O((\log n)^3(\log\log n)^2)$.

Algorithm 3: Algorithm to compute a minimal 2-3-5 chain.

Input: Integer $n > 0$.
Output: Minimal 2-3-5 chain $\mathcal{C}$ for n.

$C_{i,j} \leftarrow \emptyset$, $\overline{C}_{i,j} \leftarrow \emptyset$ for every i, j, all with length NULL
$C_{0,0} \leftarrow 0$, $a_5 \leftarrow n$, $c_5 \leftarrow NULL$
  $[k = 0]$
while $a_5 > 0$ **do**
  $a \leftarrow a_5$, $c \leftarrow c_5$, $j \leftarrow 0$, $b \leftarrow NULL$
  while $a > 0$ **do**
    for $i \leftarrow 0$ ***to*** size(a) **do**
      $a_{i-1} \leftarrow (i-1)$-th bit of a, $a_i \leftarrow i$-th bit of a
      $a_{i+1} \leftarrow (i+1)$-th bit of a, $a_{i+2} \leftarrow (i+2)$-th bit of a
      $b_i \leftarrow i$-th bit of b, $b_{i+1} \leftarrow (i+1)$-th bit of b
      $c_i \leftarrow i$-th bit of c, $c_{i+1} \leftarrow (i+1)$-th bit of c, $c_{i+2} \leftarrow (i+2)$-th bit of c
      $\mathcal{P}_{i,j} \leftarrow \emptyset$, $\overline{\mathcal{P}}_{i,j} \leftarrow \emptyset$ $[\mathcal{P}_{i,j} = \mathcal{P}_{i,j,k}$, $\overline{\mathcal{P}}_{i,j} = \overline{\mathcal{P}}_{i,j,k}]$
      Use Steps 8 to 13 of Algorithm 2 for moves $(i-1, j, k) \to (i, j, k)$
      Use Steps 14 to 23 of Algorithm 2 for moves $(i, j-1, k) \to (i, j, k)$
      if $a_i + c_i \equiv 0 \bmod 2$ **then**
        if $a_{i+1} + c_{i+1} \equiv 0 \bmod 2$ **then**
          if $c_i + a_{i+2} + c_{i+2} \equiv 0 \bmod 2$ **then**
            Include $\overline{C}_{i,j}$ and $-2^i 3^j 5^{k-1} + C_{i,j}$ in $\overline{\mathcal{P}}_{i,j}$
          else
            Include $C_{i,j}$ and $2^i 3^j 5^{k-1} + \overline{C}_{i,j}$ in $\mathcal{P}_{i,j,k}$
      else
        if $c_i + a_{i+1} + c_{i+1} \equiv 0 \bmod 2$ **then**
          Include $-2^i 3^j 5^{k-1} + \overline{C}_{i,j}$ in $\overline{\mathcal{P}}_{i,j}$
        else
          Include $2^i 3^j 5^{k-1} + C_{i,j}$ in $\mathcal{P}_{i,j}$
      $C_{i,j} \leftarrow$ shortest chain in $\mathcal{P}_{i,j}$ and update its length
      $\overline{C}_{i,j} \leftarrow$ shortest chain in $\overline{\mathcal{P}}_{i,j}$ and update its length
      Update the *movements array* (3-dimensional version of Section 3.1)
    $b \leftarrow a$, $j \leftarrow j + 1$
    $a \leftarrow a$ div 3, $c \leftarrow c$ div 3
  $c \leftarrow a_5$
    $[k = k + 1]$
  $a_5 \leftarrow a$ div 5
$C \leftarrow$ shortest chain in $\{C_{i,j}\}$
$\mathcal{C} \leftarrow$ retrace the steps for chain C using the *movements array*
return $\mathcal{C}$