SEARCHING FOR DIFFERENTIAL ADDITION CHAINS

DANIEL J. BERNSTEIN, JOLIJN COTTAAR, AND TANJA LANGE

Abstract. The literature sometimes uses slow algorithms to find minimum-length continued-fraction differential addition chains to speed up subsequent computations of multiples of points on elliptic curves. This paper introduces two faster algorithms to find these chains. The first algorithm prunes more effectively than previous algorithms. The second algorithm uses a meet-in-the-middle approach and appears to have a limiting cost exponent below 1.

1. Introduction

Fix a positive integer n. Consider the problem of computing nP given a point P on an elliptic curve defined over a finite field k. The operation counts below assume that k has odd characteristic.

One traditional solution uses an addition chain for n. Another traditional solution uses a *differential* addition chain for n starting from $0, 1$. Differential addition chains are addition chains where every addition is accompanied by a difference: one is allowed to add an entry to the chain only if the entry has the form $a + b$ where a, b, and the difference $a - b$ are already in the chain.

The differential addition chain in the second solution is typically longer than the addition chain in the first solution. The compensating advantage of the second solution is that the corresponding elliptic-curve computation can save time by working with just one coordinate of each point.

Quantitatively, Montgomery pointed out in [16] that, for elliptic curves of the form $By^2 = x^3 + Ax^2 + x$ (now called Montgomery curves), one can compute $x((a + b)P)$ from $x(aP)$, $x(bP)$, and $x((a - b)P)$ using just 6 multiplications in k if all x-coordinates are written as fractions, or 5 multiplications in k in the special case of affine difference (meaning that $x((a - b)P)$ is represented with denominator 1), or 4 multiplications in k in the doubling case $a = b$, or 3 multiplications in k for doubling an affine input. (This is not counting multiplications by constants; the doublings also involve a multiplication by the constant $(A + 2)/4$.)

A particularly easy differential addition chain to compute, given n, is a ladder. The ladder for n obtains n by doubling $n/2$ if n is even, or by adding $(n + 1)/2$ to $(n - 1)/2$ if n is odd, with $n/2$ or $(n + 1)/2, (n - 1)/2$ obtained in the same way recursively. The Montgomery ladder, meaning a ladder applied to a Montgomery

Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. Research supported in parts by the Intel Crypto Frontiers Research Center; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy–EXC 2092 CASA–390781972 "Cyber Security in the Age of Large-Scale Adversaries", the Taiwan's Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP), the Academia Sinica Grand Challenge Seed Project AS-GCS-113-M07, and the Dutch Research Council (NWO) through project 613.009.144. Permanent ID of this document: 9ee68dfacdc7d86123ed267e27eb08e7b6a740b8. Date: 2024.06.09.

Submitted to Algor. Num. Th. Symp.

curve, computes $x(nP)$ from $x(P)$ using just 9 multiplications in k per bit of n. The input $x(P)$ is in affine form, and the output $x(nP)$ is a fraction.

For comparison, the usual addition chains (or addition-subtraction chains) to compute nP from P involve one doubling per bit of n, plus some additions (or subtractions) depending on n. The number of additions per bit is $o(1)$ as $n \to \infty$, and the Bernstein–Lange formulas [5] for doubling on an Edwards curve cost just 7 multiplications in k, so a traditional addition chain on an Edwards curve costs $(7 + o(1)) \log_2 n$ multiplications in k as $n \to \infty$, outperforming the Montgomery ladder—but this does not say anything about any fixed value of n. As a numerical example, consider $n = 29$:

- The addition chain $1, 2, 3, 6, 7, 14, 28, 29$ has 4 doublings and 3 further additions. Additions are a major contributor to the cost of computing $29P$ via this chain. Even if one reduces the cost of each addition to 8 multiplications in k using the Hisil–Wong–Carter–Dawson formulas [14] for addition of an affine point on a twisted Edwards curve with twist -1 and does the first (affine) doubling in 6 multiplications, this requires 51 multiplications.
- The addition-subtraction chain $1, 2, 3, 4, 8, 16, 32, 29$, with 5 doublings and 2 further additions/subtractions, is only marginally less expensive at 50 multiplications.
- The ladder $0, 1, 2, 3, 4, 7, 8, 14, 15, 29$ is longer but, on a Montgomery curve, involves only $3 + 5 + 4 + 5 + 4 + 4 + 5 + 5 = 35$ multiplications in k.

Furthermore, comparing to ladders is not the same as comparing to arbitrary differential addition chains. Montgomery gave a "CFRC" algorithm in [17] to find a differential addition chain with, at least experimentally, approximately 1.5 additions per bit of n, while a ladder for n has 2 additions (including 1 doubling) per bit of n (for odd n).

Define $L(n)$ as the minimum length of a differential addition chain for n. The best lower bound available on $L(n)$ for most values of n is about $1/\log_2((1+\sqrt{5})/2) \approx 1.44042$ additions per bit of n. More precisely, Montgomery showed in [17, Theorem 7] that if $n = p_1 p_2 \cdots p_j$, where $p_1, p_2, \ldots, p_j$ are primes, then $n \leq 2^{j-1} F_{L(n)-j+3}$, where F is the Fibonacci sequence. Montgomery reported in [17, Table 5] that the total number of additions of ladders for small primes n was 30.3% more than this lower bound, while CFRC was just 2.0% more than this lower bound.

The chain produced by CFRC for n is a minimum-length "continued-fraction differential addition chain" for n; see Section 2 for definitions. Compared to ladders, a disadvantage of continued-fraction differential addition chains is that most of their additions on Montgomery curves use 6 multiplications in k, rather than 4 or 5. For example, $1, 2, 3, 5, 8, 13, 21, 29$ is a continued-fraction differential addition chain involving $3 + 5 + 5 + 6 + 6 + 6 + 6 = 37$ multiplications in k, slightly above the 35 multiplications in k for a ladder. Notice, however, that $6 \cdot 1.44042 \approx 8.64$ is below 9, so one would expect chains that come close enough to 1.44 additions per bit to use fewer multiplications in k than a ladder. Also, in situations where the input is not provided in affine form, a ladder incurs the cost of either an initial inversion or, for each bit of n, an extra multiplication in k.

If n is even then, given the relative cost of doublings, it is beneficial to write $n = n'2^t$, with n' odd, and use the differential addition chain to compute $n'P$ followed by t doublings to compute nP. The algorithms in this paper do not

require n to be odd, but we focus on odd n, and more specifically prime n, for numerical examples.

1.1. **Caveat regarding multiplication counts.** Counting multiplications in k is only a first step towards understanding performance of elliptic-curve operations.

Montgomery's doubling formulas cost $2M + 2S + 1C + 4A$, where M, S, C, A are the costs in k of multiplication, squaring, multiplication by a curve constant, and addition respectively. Montgomery's differential-addition formulas cost $4M + 2S + 6A$, or $3M + 2S + 6A$ for affine difference. A ladder costs $5M + 4S + 1C + 8A$ per bit since $2A$ turn out to be redundant, while 1.44 differential additions cost $5.76M + 2.88S + 8.64A$. One then wants to know whether saving $1.12S + 1C$ outweighs losing $0.76M + 0.64A$.

Switching to y^2 coordinates for Edwards curves with square d (with the 2009 Bernstein–Kohel–Lange formulas; see [4]) changes $2M + 2S + 1C + 4A$ to $4S + 2C + 4A$, changes $4M + 2S + 6A$ to $4M + 2S + 1C + 6A$, changes $3M + 2S + 6A$ to $3M + 2S + 1C + 6A$, and changes $5M + 4S + 1C + 8A$ to $3M + 6S + 3C + 8A$. One then wants to know whether $3.12S + 1.56C$ outweighs $2.76M + 0.64A$.

One can find papers that measure costs of operations, for example reporting $S \approx 0.8M$ and $A \approx 0.1M$ for particular fields k. In that example, the cutoff for C/M is about 0.072 for Montgomery curves, and about 0.21 for Edwards curves with square d. Smaller values of C/M favor ladders, while larger values of C/M favor shorter differential addition chains. If the 1.44 changes then these numbers change correspondingly.

All of the numbers change again if one moves beyond elliptic curves to higher dimensions, as in, e.g., [10].

The rest of this paper focuses on the lengths of differential addition chains, and leaves lower-level performance analysis to future work.

1.2. **Existing algorithms to search for differential addition chains.** CFRC builds a differential addition chain for n by choosing an auxiliary integer ρ coprime to n and applying Euclid's gcd algorithm to the pair (n, ρ). Euclid's algorithm is not instantaneous: the CFRC algorithm stated in [17] uses $\Theta((\log n)^2)$ subtractions for an average ρ. (See [23] for more precise bounds.) One can instead compute the length of the (n, ρ) chain by carrying out $O(\log n)$ divisions using $O((\log n)^2)$ bit operations, or, using [18] and [13], asymptotically $O((\log n)(\log \log n)^2)$ bit operations. But the big issue is that there are $\Theta(n)$ choices of ρ to check.

Montgomery in [17, Section 7] proposed a much faster algorithm, PRAC, to find a chain for n, but the chains produced by PRAC have disadvantages compared to the chains produced by CFRC:

- PRAC generates a differential addition-subtraction chain, but not always a differential addition chain.
- Even when it generates a differential addition chain, PRAC usually does not generate a continued-fraction chain. Applying a PRAC chain uses more complicated data flow and more storage than applying a continued-fraction chain: compare [17, Table 4, "Action(s)" column] to Algorithm 1.
- Banegas, Bernstein, Campos, Chou, Lange, Meyer, Smith, and Sotáková pointed out as part of "CTIDH" [2] that one can, at low cost, hide the exact value of n in a continued-fraction chain, for contexts where that

value is secret. Converting the evaluation of a PRAC chain to a constant-time computation would incur higher costs. See [2, Section 2.4] for further information on constant-time computations.

- Even in situations where all that matters is the number of additions, PRAC finds chains that are not quite as short as the CFRC chains. For example, [17, Table 5] says 7.7% longer than the lower bound for primes $n < 10^6$. See also the figures in Section 5 of this paper.

A slower variant of PRAC suggested in [17, Section 8] addresses the last disadvantage but not the others.

Cervantes-Vázquez, Chenu, Chi-Domínguez, De Feo, Rodríguez-Henríquez, and Smith reported in [7] that for isogeny computations they had "pre-computed the shortest differential addition chains" for all small primes—actually minimum-length *continued-fraction* differential addition chains, like CFRC. The SIBC [1] library from Adj, Chi-Domínguez and Rodríguez-Henríquez includes the software for this precomputation. The chain-search algorithm is different from CFRC, with some advantages and some disadvantages; see Section 3.1.

The CTIDH software from [2] includes improvements to the search algorithm from [7]. For both [7] and [2], the range of primes is small enough that the search is easily affordable in any case, but larger primes naturally appear in other isogeny-based computations; see, e.g., [11]. It is thus interesting to consider how quickly one can find the shortest continued-fraction differential addition chain for a given n, and, more broadly, what tradeoffs are possible between chain-searching time and chain length.

1.3. **Contributions of this paper.** This paper introduces two new algorithms that map any given integer $n \geq 3$ to a minimum-length continued-fraction differential addition chain for n.

Our first algorithm, like the previous algorithms from [7] and [2], is a low-memory combinatorial search.[1] The advantage of this algorithm is that it prunes the search tree more effectively than the previous algorithms. Experiments suggest that the number of search nodes for an average input n is bounded by a small constant times n, unlike the previous algorithms.

Here is a numerical example to illustrate what this algorithm achieves. Consider the prime divisor $n = 358429848460993 \approx 2^{48.35}$ of $2^{448} - 1$. A search through $227081419196942 \approx 2^{47.69}$ nodes, using in total 13319 core-minutes (about 8 core-cycles per node) on a dual AMD EPYC 7742, found a 73-addition continued-fraction differential addition chain ending in ρ, n where $\rho = 221521718594876$. This ρ determines the entire chain and encodes the expensive part of the search. To construct the full chain, simply apply Euclid's subtractive algorithm to (n, ρ).

For comparison, the algorithm from [2] would have been thousands of times slower. The CFRC approach of searching through all possible values of ρ would, with some added pruning, be faster than the algorithm from [2], but would still involve many more than n nodes.

Our second algorithm is a meet-in-the-middle search for which the only obvious bottleneck is searching $n^{2/3+o(1)}$ nodes. One might even hope for a meet-in-the-middle search to reach exponent $0.5+o(1)$, but the meet condition in this algorithm

[1] Actually, the software from [7] accumulates a list of all nodes in the search tree and is thus not low-memory, but one can trivially eliminate this list.

is more complicated than an equality test. A full analysis of this algorithm is difficult, but we find many values of $n < 10^9$ for which this algorithm outperforms the first algorithm, and the trends appear consistent with the idea that the limiting exponent for the second algorithm is below 1.

This algorithm has a disadvantage: the algorithm's memory usage grows similarly to the number of nodes. It is not clear whether the lower-memory "golden collision" techniques of [22] are applicable here.

Finally, we point out that generating short continued-fraction differential addition chains for a large range of n is more efficient than generating chains for one n at a time. In our meet-in-the-middle search, the "left" part of the search can be shared; see Section 4.7. In the extreme case that one wants to generate chains for every small prime (which is essentially the situation in [7] and [2]), one can enumerate all short chains as mentioned by Montgomery in [17, Section 6], but experiments and heuristics suggest that it suffices to focus on chains having only a small fraction of swaps.

1.4. **Searches for other types of addition chains.** There is a much longer history of searches for addition chains without the constraint of being differential addition chains. For systematic search algorithms, see Chin–Tsai [8], Bleichenbacher–Flammenkamp [6], Thurber [20], Knuth [15], Clift [9], and Thurber–Clift [21]. The algorithms from [15] and [9] reuse work across all small values of n.

One might expect searches for the shortest addition chains to be more difficult than searches for the shortest continued-fraction differential addition chains: there are many more choices of what to add at each step, even when known symmetries are taken into account. On the other hand, a search for a shortest addition chain for n can be pruned at a shorter length, and in particular at the length of an easy-to-find addition chain for n having only slightly more than $\log_2 n$ steps. The number of nodes visited in the search from [6] is plotted in [6, Figure 8.1] for n ranging from 2^{11} to 2^{19} and shows a wide variance on a logarithmic scale for each size of n, with upper outliers ranging from about 2^7 to about 2^{22} while most values are much smaller. The average-case and worst-case exponents of the search are unclear.

The literature considers further types of chains. For example, [12] uses small-discriminant complex multiplication to efficiently reduce single-scalar multiplication $n, P \mapsto nP$ to double-scalar multiplication $m, n, P, Q \mapsto mP + nQ$ with half-size scalars m, n. Even without such curve structure, sometimes applications naturally involve double-scalar multiplication, triple-scalar multiplication, etc. Differential addition chains for (m, n) are considered in, e.g., [19, Chapter 3] and [3]; it would be interesting to investigate algorithms to find such chains of minimum length.

2. Preliminaries

An **addition chain** for an integer n is defined as a sequence of integers

$$1 = c_0, c_1, \ldots, c_r = n$$

such that, for each $i \in \{1, \ldots, r\}$, there exist $j, k \in \{0, \ldots, i-1\}$ such that $c_i = c_j + c_k$.

To formalize differential addition chains for n as special cases of addition chains for n, we also start differential addition chains with 1. A **differential addition chain** for n is then defined as a sequence of integers $1 = c_0, c_1, \ldots, c_r = n$ such that, for each $i \in \{1, \ldots, r\}$, there exist $j, k \in \{0, \ldots, i-1\}$ such that $c_i = c_j + c_k$

and $c_j - c_k \in \{0, c_0, c_1, \ldots, c_{i-1}\}$. In other words, for each addition the difference is already in the chain or the difference is 0.

An example of a differential addition chain for $n = 29$ is $1, 2, 3, 5, 8, 13, 21, 29$, where for example when $13 = 8 + 5$ is added to the chain, we see that the difference $8 - 5 = 3$ is already in the chain.

A **continued-fraction tuple sequence** for $n \geq 3$ is defined as a sequence of tuples of three integers $(a_2, b_2, c_2), \ldots, (a_r, b_r, c_r)$ satisfying the following conditions: $(a_2, b_2, c_2) = (1, 2, 3)$; $c_r = n$; and, for each $i \in \{3, \ldots, r\}$, one has $(a_i, b_i, c_i) = (b_{i-1}, c_{i-1}, c_{i-1} + b_{i-1})$ or $(a_i, b_i, c_i) = (a_{i-1}, c_{i-1}, c_{i-1} + a_{i-1})$. Notice that each tuple satisfies $c_i = a_i + b_i$.

For example, the tuples

$$(1, 2, 3), (2, 3, 5), (3, 5, 8), (5, 8, 13), (8, 13, 21), (8, 21, 29)$$

form a continued-fraction tuple sequence for $n = 29$. In this example, one has $(a_i, b_i, c_i) = (b_{i-1}, c_{i-1}, b_{i-1} + c_{i-1})$ for the second, third, fourth, and fifth tuples, and $(a_i, b_i, c_i) = (a_{i-1}, c_{i-1}, a_{i-1} + c_{i-1})$ for the sixth tuple.

A **continued-fraction differential addition chain** for $n \geq 3$ is defined as a chain of the form $1, 2, c_2, \ldots, c_r = n$ where $(a_2, b_2, c_2), \ldots, (a_r, b_r, c_r)$ is a continued-fraction tuple sequence for n. In other words, the chain is $1, 2$ followed by the last entry in each tuple. To see that this is, in fact, a differential addition chain, notice that each a_i and each b_i is in the chain, and that each c_i is obtained as $c_{i-1} + b_{i-1}$ with difference $c_{i-1} - b_{i-1} = a_{i-1}$ already in the chain, or as $c_{i-1} + a_{i-1}$ with difference $c_{i-1} - a_{i-1} = b_{i-1}$ already in the chain.

For example, the above differential addition chain $1, 2, 3, 5, 8, 13, 21, 29$ adheres to this constraint and is thus a continued-fraction differential addition chain for $n = 29$.

Note that $1, 2, 3, 5, 8, 13, 16, 29$ is also a valid differential addition chain for $n = 29$, but it does not adhere to the continued-fraction constraints. Specifically the 16 is not valid, since it cannot be created from $(5, 8, 13)$ following the constraint. This paper focuses on continued-fraction chains.

2.1. **Compression.** A continued-fraction tuple sequence is completely determined by one bit at each position:

- Bit $f_i = 0$ means $(a_i, b_i, c_i) = (b_{i-1}, c_{i-1}, c_{i-1} + b_{i-1})$.
- Bit $f_i = 1$ means $(a_i, b_i, c_i) = (a_{i-1}, c_{i-1}, c_{i-1} + a_{i-1})$.

One can thus compress a length-r continued-fraction differential addition chain $1, 2, c_2, \ldots, c_r$ to just $r - 2$ bits $f_3, \ldots, f_r$. This compressed format is convenient for Algorithm 1 below.

2.2. **Evaluating continued-fraction differential addition chains.** Given a point P and a continued-fraction differential addition chain (in compressed form) for n, Algorithm 1 computes nP. This algorithm uses a subroutine DBL, which doubles a point P efficiently, and a subroutine dADD, which efficiently maps $Q - P, P, Q$ to $Q + P$. These algorithms depend on the type of elliptic curves used; for efficient algorithms see [4].

Despite the fact that differential addition chains are usually longer than binary addition chains, we need significantly fewer computations per bit in the compressed chain. This algorithm only takes one doubling and one addition as a setup, followed with one addition per bit of the chain, while the binary addition chain requires

Algorithm 1: Multiplication-by-n map using a differential addition chain

Input : a point P, a compressed continued-fraction differential addition chain $f_3, \ldots, f_r$ for n

Output: nP

$X_0 \leftarrow P$
$X_1 \leftarrow \texttt{DBL}(P)$
$X_2 \leftarrow \texttt{dADD}(P, P, X_1)$
for $i = 3$ **up through** r **do**
  if $f_i = 0$ **then**
    $(X_0, X_1, X_2) \leftarrow (X_1, X_2, \texttt{dADD}(X_0, X_1, X_2))$
  else
    $(X_0, X_1, X_2) \leftarrow (X_0, X_2, \texttt{dADD}(X_1, X_0, X_2))$
return X_2

one doubling and possibly one addition per bit of the chain. Furthermore, the differential case works with fewer coordinates of each point. See Section 1.

A different way to express the inner loop in Algorithm 1 is as follows: swap X_0 with X_1 if $f_i = 1$; then set $(X_0, X_1, X_2) \leftarrow (X_1, X_2, \mathrm{dADD}(X_0, X_1, X_2))$. In situations where the exact value of n (as opposed to the chain length r) is secret, one should carry out the conditional swap in time independent of f_i, hiding whether there was in fact a swap.

2.3. **Bounds on the length.** Montgomery's bound on the length of differential addition chains is particularly easy to prove for the special case of continued-fraction differential addition chains, and does not require any assumptions on the factorization of n: for every integer $n \geq 3$, one has $n \leq F_{L^{***}(n)+2}$, where $L^{***}(n)$ is the minimum length of a continued-fraction differential addition chain for n and F is again the Fibonacci sequence.

Indeed, say $(a_2, b_2, c_2), \ldots, (a_r, b_r, c_r)$ is a minimum-length continued-fraction tuple sequence for n. By induction $a_i \leq F_i$, $b_i \leq F_{i+1}$, and $c_i \leq F_{i+2}$ for each i. (For $i = 2$, one has $a_2 = 1 = F_2$, $b_2 = 2 = F_3$, and $c_2 = 3 = F_4$. For $i \geq 2$, one has $a_{i-1} \leq F_{i-1}$ and $b_{i-1} \leq F_i$ and $c_{i-1} \leq F_{i+1}$ by the inductive hypothesis; and then $(a_i, b_i, c_i) = (b_{i-1}, c_{i-1}, c_{i-1} + b_{i-1})$, in which case $a_i \leq F_i$ and $b_i \leq F_{i+1}$ and $c_i \leq F_{i+1} + F_i = F_{i+2}$ as desired, or $(a_i, b_i, c_i) = (a_{i-1}, c_{i-1}, c_{i-1} + a_{i-1})$, in which case $a_i \leq F_{i-1} \leq F_i$ and $b_i \leq F_{i+1}$ and $c_i \leq F_{i+1} + F_{i-1} \leq F_{i+2}$ as desired.) In particular, $n = c_r \leq F_{r+2}$.

This bound is tight when n is a Fibonacci number. This corresponds to a compressed differential addition chain with $r - 2$ zeros, since then for each step the two largest elements of the tuple are added. For example, the sequence

$$(1, 2, 3), (2, 3, 5), (3, 5, 8), (5, 8, 13), (8, 13, 21)$$

and corresponding chain $1, 2, 3, 5, 8, 13, 21$ reach $n = 21$ with $r = 6$ additions; 21 is indeed the 8th Fibonacci number.

The explicit formula $F_i = (\phi^i - \psi^i)/\sqrt{5}$, where $\phi = (1 + \sqrt{5})/2 = 1.618\ldots$ and $\psi = (1 - \sqrt{5})/2 = -0.618\ldots$, implies $F_{r+2} < (\phi^{r+2} + 0.24)/\sqrt{5}$, which in turn implies $L^{***}(n) = r > \log_\phi(n\sqrt{5} - 0.24) - 2 > 1.44042 \log_2(n - 0.11) - 0.32773$.

For a trivial upper bound on $L^{***}(n)$, observe that the chain $1, 2, 3, 4, 5, \ldots, n$ that repeatedly adds 1 is a continued-fraction differential addition chain, namely the all-swap chain of length $r = n - 1$ that reaches n via tuples $(1, 2, 3)$, $(1, 3, 4)$, $(1, 4, 5)$, etc. This is not a useful upper bound in any shape or form.

Montgomery conjectures in [17, Section 6] that each $n \geq 3$ has a continued-fraction differential addition chain of length at most $1.77 \log_2 n + O(1)$. Montgomery notes that the actual minimum length for primes $n < 10000$ is below $1.5 \log_2 n + 1$ except for $n \in \{3847, 5903\}$; our computations found more exceptions, such as $n = 1540631$ (length $32 \approx 1.5 \log_2 n + 1.167$), $n = 64398343$ (length $40 \approx 1.5 \log_2 n + 1.089$), and $n = 160984639$ (length $42 \approx 1.5 \log_2 n + 1.106$).

3. Pruned left-to-right combinatorial searches

This section reviews the algorithms from the SIBC [1] and CTIDH [2] software, and presents our first algorithm as an improvement to the algorithm from [2].

3.1. **Baseline search.** As a simplified variant of what the SIBC software does, consider the following brute-force enumeration of all continued-fraction differential addition chains of length at most $2 + \lfloor 1.5 \log_2 n \rfloor$.

The chains are enumerated as nodes in a search tree. A length-r chain for $r \geq 2$ is represented as $r - 2$, the tuple (a_r, b_r, c_r), and an $(r - 2)$-bit integer whose bits are the chain in compressed form (see Section 2.1). The root node has $r = 2$ and tuple $(1, 2, 3)$. Each node below the maximum length has two child nodes for the two choices of $(a_{r+1}, b_{r+1}, c_{r+1})$, so overall there are $2^{1+\lfloor 1.5 \log_2 n \rfloor} - 1 \approx n^{1.5}$ nodes.

It seems that, after enumerating all of the chains, one can always take a minimum-length chain that ends with n. No examples are known for which this search fails, i.e., for which none of the chains end with n.

The basic advantage of this algorithm over Montgomery's CFRC is that there are just a few arithmetic operations per chain. The basic disadvantage is that the number of chains considered is larger, at least asymptotically. The point of the pruning techniques covered below is to reduce the number of chains considered.

3.2. **Pruning nodes that overshoot the target.** The search that is actually in the SIBC software differs in one way from the above baseline search: if a node has $c_r > n$, then the node is pruned, meaning that all descendants of the node are eliminated.

Experiments show that this pruning leaves most of the nodes: i.e., most nodes have c_r much smaller than n. For example, for $n = 6521$, this pruning leaves more than 90% of the $2^{20} - 1$ nodes.

To understand why this happens, consider an all-0 compressed chain (no swaps), i.e., a Fibonacci chain, increasing by a factor approximately 1.618 at each step, so about $1.5 \log_2 n$ steps reach about $n^{1.04}$. One can check that inserting a swap divides the end of the chain by a factor close to 1.118, in effect reducing one 1.618 factor to 1.447, although the factor can be noticeably larger or smaller if the inserted swap is close to the end of the chain. A back-of-the-envelope estimate is then that s swaps reduce the end of the chain from about $n^{1.04}$ to about $n^{1.04}/1.1^s$, dropping below n once s reaches about $0.3 \log_2 n$. An average chain, before pruning, has about $0.75 \log_2 n$ swaps, so presumably its end is much smaller than n.

Example: Figure 1 displays the tree of tuples considered in the SIBC software's recursive search for a continued-fraction chain for $n = 11$. The root of the tree is

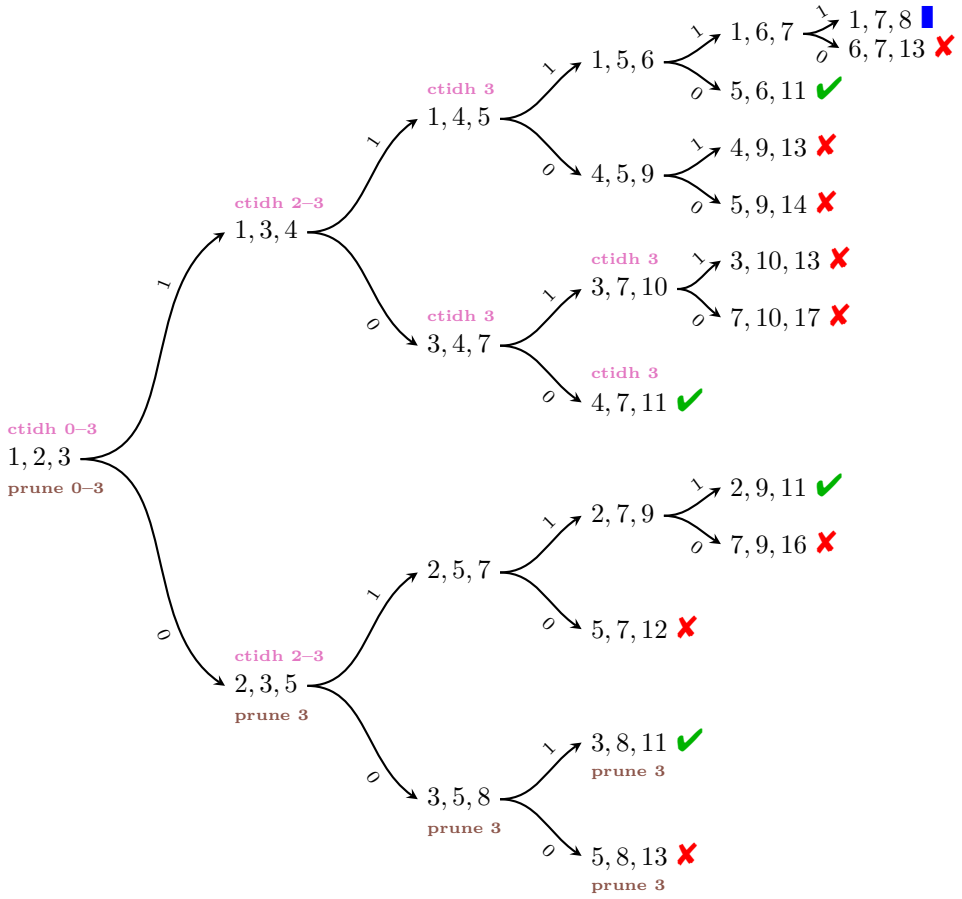

FIGURE 1. Chains considered by the SIBC search, the CTIDH search, and this paper's pruning algorithm for $n = 11$. Each node is labeled with the tuple $c - b, b, c$ where the chain at the node has the form $\ldots, b, c$. See text for further description.

the chain $(1, 2, 3)$. Each chain in the tree has at most $\lfloor 1.5 \log_2 11 \rfloor = 5$ additions, apart from the two to create our starting tuple $(1, 2, 3)$. The chains with a red cross (✘) have overshot the target $n = 11$, while the chain with a blue vertical rectangle (▮) has reached the maximum number of additions, while staying strictly beneath the target integer. The chains with a green checkmark (✔) are considered since they end at $n = 11$ and they are within the length limit. The SIBC software then simply chooses one of the chains for n of minimum length.

3.3. **An incremental length search.** One change from the SIBC search to the CTIDH search is that the CTIDH search first tries chain length 0, then chain length 1, then chain length 2, etc. As soon as a chain ending in n is found it is for sure (one of) the shortest chain(s), so the search immediately stops, saving all of the time for the other chains of that length and saving all of the time for longer chains, rather than continuing to find all chains up through the maximum length.

This also eliminates concern about the possibility of the precomputed length bound $2 + \lfloor 1.5 \log_2 n \rfloor$ being too small.

A disadvantage is that chains encountered for any particular length are repeating chains that were already built for earlier lengths. One expects this to lose only a small constant factor. It would be possible, at the expense of memory, to save the previously built chains rather than rebuilding them.

3.4. **Pruning nodes that undershoot the target.** There is one other change from the SIBC search to the CTIDH search: nodes that are clearly too small are pruned.

The test for nodes being too small considers doublings until the target length r: a node with tuple (a_i, b_i, c_i) is pruned if $2^{r-i}c_i < n$. If even with doublings we do not reach the target then clearly all descendants of the node will undershoot. This test relies on the target length being known; see Section 3.3.

An adaptation of the argument from Section 3.2 suggests that this pruning eliminates most nodes. The impact was not analyzed in [2], but our experiments show that the CTIDH search is much faster than the SIBC search; see Section 5.

Example: Figure 1, already described in Section 3.2 as showing the tree of chains considered in the SIBC search for the target $n = 11$, also shows the tuples considered by the CTIDH search for the same target. The "ctidh" labels show which tuples are considered by this search. Specifically, a label "ctidh 3" indicates a tuple considered by the CTIDH search for length 3, meaning 3 additions after the starting chain $1, 2, 3$; a label "ctidh 2–3" indicates a tuple considered for length 2 or 3; a label "ctidh 0–3" indicates a tuple considered for length 0, 1, 2, or 3. In more detail:

- For length 0 or 1, the search will not make it past the first tuple $(1, 2, 3)$, since doubling 0 or 1 times leads to 3 or 6 respectively, which both undershoot the target $n = 11$.
- For length 2, we make it to $(1, 3, 4)$ and $(2, 3, 5)$, but doubling 4 and 5 once will both lead to undershooting again.
- For length 3, we find $(1, 4, 5)$ in the third column (two additions after the root); doubling 5 again undershoots. We then try $(3, 4, 7)$ in the third column; this does not undershoot. So we continue with another addition. This leads to the tuples $(3, 7, 10)$, which undershoots since we are at the maximum number of additions, and $(4, 7, 11)$, which reaches the target.

At this point the search has found the continued-fraction chain $1, 2, 3, 4, 7, 11$, which is guaranteed to be of minimum length. The depth-first search can stop at this point but, imitating [2], actually continues at this point with a lower length limit, so it visits $(2, 3, 5)$ again; this is why $(2, 3, 5)$ is labeled "ctidh 2–3" in Figure 1 rather than "ctidh 2".

3.5. **A more effective test for undershooting the target.** Our first new algorithm is based on the CTIDH algorithm. It reuses the idea of searching lengths incrementally until a chain is found, stopping the algorithm as soon as a chain is found. A small difference here is that we use a non-recursive depth-first search through the available graph (based on the length we are considering at that moment), considering 0 before 1. The critical difference though is that the new algorithm prunes more nodes that are guaranteed to undershoot the target.

There are no doublings in a continued-fraction differential addition chain after the initial $1, 2$. If a chain of length i reaches tuple (a_i, b_i, c_i), then a child chain of

length $i+1$ reaches at most $2c_i$ (since $b_i \le c_i$), a descendant chain of length $i+2$ reaches at most $3c_i$, a descendant chain of length $i+3$ reaches at most $5c_i$, etc. Note the Fibonacci sequence here.

For any particular target length r, we precompute a list of bounds that the chain at a certain length needs to have surpassed or the chain will clearly undershoot. Specifically, if $c_r \le n-1$ then c_r undershoots; if $c_{r-1} \le \lfloor (n-1)/2 \rfloor$ then c_{r-1} undershoots; if $c_{r-2} \le \lfloor (n-1)/3 \rfloor$ then c_{r-2} undershoots; if $c_{r-3} \le \lfloor (n-1)/5 \rfloor$ then c_{r-3} undershoots; etc. We then prune any nodes not meeting these conditions.

It would be interesting to explore further pruning possibilities. We could prune more nodes by accounting for b_i usually being noticeably smaller than c_i, although this would involve more arithmetic. Also, the test for overshooting could account for at least a_i being added to c_i at each subsequent step. Improvements to the overshooting test could have more impact than in Section 3.2 relative to the total number of nodes, since many nodes that undershoot have disappeared.

Example: To illustrate how our pruning algorithm works, we will again look at the same target as the other examples, $n = 11$. We start with incrementing lengths, but now per length we will first compute the list of Fibonacci-based bounds $\ldots, \lfloor (n-1)/5 \rfloor, \lfloor (n-1)/3 \rfloor, \lfloor (n-1)/2 \rfloor, n-1$:

- For length 0, the list of bounds is $3, 5, 10$. The first (and only) chain considered here is $1, 2, 3$ and since $3 \le 10$ this will clearly undershoot.
- For length 1, the list of bounds is $2, 3, 5, 10$ and again the search will immediately stop at the root chain $1, 2, 3$, since $3 \le 5$.
- For length 2, the list of bounds is $1, 2, 3, 5, 10$ and again the search stops since $3 \le 3$. Notice the improvement here compared to the CTIDH search.
- For length 3, the list of bounds is $0, 1, 2, 3, 5, 10$. We will now make it past the first step since $3 > 2$. Then $2, 3, 5$ is not pruned since $5 > 3$. The next tuple considered (since we consider child 0 before child 1) is $3, 5, 8$, which again is not pruned since $8 > 5$. Then the next tuple $5, 8, 13$ is pruned since 13 overshoots. The final tuple considered is $3, 8, 11$ which then results in the chain $1, 2, 3, 5, 8, 11$.

The nodes considered here are labeled "prune" in the tree of tuples in Figure 1.

4. Meet-in-the-middle algorithm

This section presents our second algorithm. This algorithm uses a conceptually simple meet-in-the-middle (MitM) approach to search for a continued-fraction differential addition chain for a given integer $n \ge 3$.

This section actually presents two variants of the algorithm: a "left-interval" variant and a "left-length" variant. The left-interval variant does not always find minimum-length continued-fraction chains, although experiments show that it usually does. The left-length variant always finds minimum-length continued-fraction chains. The abstract and introduction of this paper are referring to the left-length variant.

4.1. Chain structure.

This algorithm views a short chain in compressed form (see Section 2.1) as the concatenation of a left side and a right side. Enumerating, e.g., $n^{2/3+o(1)}$ possibilities for the left side and $n^{1/3+o(1)}$ possibilities for the right side is tantamount to creating a space of $n^{1+o(1)}$ chains overall.

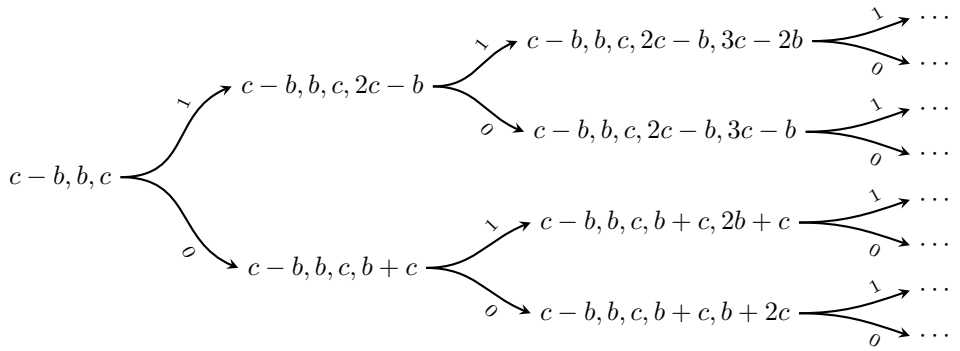

FIGURE 2. Tree of all possible right-side chains, viewed as chains of elements of $\mathbb{Z}[b,c]$ starting with $c-b,b,c$.

The two basic challenges addressed in this section are, first, to create short left and right sides having appropriate sizes for the outputs of the concatenated chains to be "random" integers around n, making it plausible that a space of $n^{1+o(1)}$ chains will include a chain for n; and, second, to identify matching left-side and right-side possibilities without writing down most of the $n^{1+o(1)}$ chains.

4.2. **Algorithm structure.** The left side of this algorithm is straightforward. We start with $(1,2,3)$ and we look at all the chains that end at a particular cutoff. The cutoff depends on the algorithm variant and on n:

- In the left-interval variant, we enumerate chains whose final entry is in an interval chosen based on n. The function mapping n to an interval is an algorithm parameter. For example, one can take the interval from $\lceil n^{2/3} \rceil$ to $\lceil 2n^{2/3} \rceil$.
- In the left-length variant, we instead choose a length and enumerate chains of that length. For example, if the goal is to find a chain for n of total length ℓ, one can choose length $\lfloor 2\ell/3 \rfloor$ for the length of the left side of the chain. (The algorithm has an outer loop searching incrementally through ℓ, as in Section 3.3.)

With either variant, the enumeration of left-side chains uses a simple generalization of the algorithm from Section 3; see Section 4.4 for details.

The right side and meet-in-the-middle would be very simple as well if we could use the right bits in a compressed chain to work backwards from n to an intermediate point. However, we only know at the right side that the chain ends on n; we do not know any of the previous steps. CFRC addresses this by guessing many possibilities for the step before n and then working entirely from right to left, but we do not want to try so many guesses.

What we do know is that, after the left side produces an ending tuple $(a,b,c) = (c-b,b,c)$ for some integers b and c, the right side takes this input and outputs $(\ldots,\ldots,pb+qc)$, where p and q are integers determined by the right side. (For example, the root chain $c-b,b,c$ in Figure 2 has $(p,q)=(0,1)$; chain $c-b,b,c,2c-b$ has $(p,q)=(-1,2)$; etc.) The concatenated chain outputs n exactly when $pb+qc$ equals n.

So we generate various right sides of reasonable sizes as explained in Section 4.6. For each right side, we have p, q such that the right side maps $(c - b, b, c)$ to $(\ldots, \ldots, pb + qc)$. Meanwhile we know (b, c) for each left side. We could now check each left side against each right side to see whether $pb + qc$ equals n, but we want to check fewer combinations.

We pick a prime modulus m. For each left side, we reduce (b, c) modulo m, obtaining an index $(b \bmod m, c \bmod m)$ for that chain. Then, for each right side, we take the corresponding (p, q) and quickly locate the indices for which the following equation holds:

$$bp + cq \equiv n \pmod{m}.$$

For each such index, and for each left side having that index, we then check whether $bp + cq = n$ also holds without the mod m. In cases where $bp + cq = n$, we concatenate the left side and right side to form a chain for n.

4.3. **Exponent intuition.** One expects the line $bp + cq \equiv n \pmod{m}$ to have about m solutions for $(b \bmod m, c \bmod m)$, except in rare cases such as $p \equiv q \equiv n \equiv 0 \pmod{m}$ (which will certainly not happen if n is a prime larger than m). An average right side will then consider m indices, which presumably will have only about $1/m$ of the left sides, saving a factor m compared to concatenating all left and right sides. Considering the m indices is not a bottleneck if the number of left sides is around m^2 (or larger).

In particular, if there are $n^{2/3+o(1)}$ left sides and $n^{1/3+o(1)}$ right sides, with $m \in n^{1/3+o(1)}$, then one expects total algorithm cost to only be $n^{2/3+o(1)}$.

4.4. **Left-hand side.** Consider first the left-interval variant. The task here is to enumerate left-side chains ending with c in a specified target interval. As in Section 3, we combinatorially enumerate partial chains as nodes in a search tree. We are free to prune nodes that will not be useful in building a short chain for n.

We first construct the range of accepted end points c as follows. We pick an exponent e with $0 < e < 1$, a factor $f > 0$, and a factor $s \geq 1$. We then use the target interval $\lceil \lceil fn^e \rceil / s \rceil$ through $\lfloor \lceil fn^e \rceil s \rfloor$.

A continued-fraction chain cannot jump by a factor more than 2 in one step, so taking $s \approx \sqrt{2}$ suffices to guarantee that any desired chain will have a left side arriving in the target interval. Our experiments take a more optimistic $s = 1.1$, falling back to a larger s only if no short chains are found; this is one reason that our experiments with the left-interval variant are not guaranteed to find the shortest chains. Our experiments take $f = 1$ and $e = 2/3$.

After choosing a target interval, we choose a maximum length for the left-side chain (length here meaning the number of additions on the left side after the initial $1, 2, 3$), specifically $\lfloor 1.5 \log_2 C \rfloor$ where C is the maximum integer in the interval. This limit on the length is another reason that the left-interval variant is not guaranteed to find the shortest chains: even if one takes s large enough to guarantee that any desired chain has a left side arriving in the target interval, there is no proof that the left side will do so within this length.

After making these choices, we carry out a combinatorial search for left-side chains, pruning as in Section 3:

- A node is pruned if the chain length exceeds the specified limit.
- A node is pruned if it will clearly overshoot the top of the target interval: specifically, if c already exceeds the top.

- A node is pruned if it will clearly undershoot the bottom of the target interval. Specifically, if a node has a chain of length k ending c, then a child node of length $k+1$ ends at most $2c$, a grandchild node of length $k+2$ ends at most $3c$, a great-grandchild node of length $k+3$ ends at most $5c$, etc., as in Section 3. We prune the node ending c if this Fibonacci chain guarantees that descendants of the node will undershoot the target.

We gather all chains that make it through this pruning process, while remembering the compressed form of the chain, the length of the chain, and the tuple (b, c).

For the left-length variant, the task is instead to enumerate left-side chains of exactly a specified length. We prune a node if the chain length goes beyond that length, and we use a Fibonacci test to prune a node if it will clearly undershoot n. We could also test for overshooting n, but this would be useful only for left-side lengths long enough to allow such overshooting.

4.5. **Dictionaries.** The next step is to create a dictionary (associative array) that allows the left sides to be accessed according to their indices $(b \bmod m, c \bmod m)$. Each key in the dictionary is one of the m^2 pairs of integers mod m, and stores a list of all left sides with that index.

4.6. **Right-hand side.** As discussed above, the right-hand side is a bit more complicated, since we only know the number we want to end up at. We enumerate right sides starting from the tuple $((-1,1),(1,0),(0,1))$ to represent $(c-b,b,c)$, followed by the tuple $((1,0),(0,1),(1,1))$ to represent $(b,c,b+c)$ or the tuple $((-1,1),(0,1),(-1,2))$ to represent $(c-b,c,2c-b)$, etc. We apply three types of pruning to right-side chains:

- We prune $(\ldots,\ldots,(p,q))$ if it would clearly have $pb+qc$ overshoot n. Specifically, we take ranges for b, c (such ranges are known from the computation of left-side chains), appropriately multiply and add to compute a lower bound on $pb+qc$ (noting that p can be negative), and prune if this lower bound is larger than n.
- We prune $(\ldots,\ldots,(p,q))$ if it would clearly have $pb+qc$ undershoot n. This applies a Fibonacci test to an upper bound on $pb+qc$. For the left-length variant, we use a test that is guaranteed to work, as in Section 3: the sequence entries after $pb+qc$ are at most $2(pb+qc)$, $3(pb+qc)$, $5(pb+qc)$, $8(pb+qc)$, etc. For the left-interval variant, we replace $2,3,5,8,\ldots$ with $1,2,3,5,\ldots$, pruning beyond what is guaranteed to work.
- We prune $(\ldots,\ldots,(p,q))$ if the right-side chain length is beyond a maximum allowed length computed as explained below.

For the left-interval variant, we first enumerate and store left-side chains (restarting with a larger s if no left-side chains are found). We then incrementally search for total chain lengths (not counting the initial $1,2,3$). For each total chain length, we subtract the minimum left-side length from the total length to obtain a maximum length of the chains that need to be considered on the right-hand side; we then enumerate right-side chains within that limit.

For the left-length variant, we incrementally search for total chain lengths as an outer loop, as noted earlier. We partition each total chain length ℓ into a left-side length ℓ_0 and a right-side length $\ell-\ell_0$; we then enumerate and store left-side chains, and enumerate right-side chains. Our experiments choose $\ell_0 = \lfloor 2\ell/3 \rfloor$.

With either variant, for each right side $(\ldots, \ldots, (p, q))$ that survives the pruning (and that has exactly length $\ell - \ell_0$ for the left-length variant), we compute all indices $(b \bmod m, c \bmod m)$ such that $pb + qc \equiv n \pmod m$. Normally this takes $\Theta(m)$ operations per right side.

For each such index, we check the dictionary for all left sides (b, c) having that index. The final check for each (b, c) is whether $p \cdot b + q \cdot c = n$.

We concatenate all left-right pairs satisfying this condition. For the left-interval variant, this often finds multiple options for chains of different lengths; we output the concatenation having minimum length. For the left-length variant, we stop as soon as one pair is found satisfying this condition, since that pair is guaranteed to have minimum length.

If no pairs satisfy this condition, we increment the total chain length and try again. For the left-interval variant, if the total chain length reaches $\lceil 1.5 \log_2 n + 1 \rceil$ then we increase s and start over; as in Sections 3.1 and 3.2, this is not guaranteed to terminate, but has not been observed to fail. The left-length variant is guaranteed to terminate with a minimum-length continued-fraction chain.

4.7. **Precomputation.** Parts of this algorithm can be precomputed when multiple primes are targeted that are close to each other. In this case the interval of the left side and the boundaries on the chains of the right side are basically the same and thus the output can be reused (or even precomputed once before starting the computations). And since the moduli will probably also be similar, or at least one can use the same modulus for all primes, the dictionary step can also be precomputed. This would mean that we have an algorithm that inputs the dictionary and the target and outputs the final chain. This greatly reduces the time needed.

Most of this paper, including all of Section 5, instead considers one target n at a time. The number of search nodes reported in Section 5 is the full number of nodes for each n, not the smaller number of nodes that would appear after a shared precomputation.

4.8. **Example.** This example focuses on the left-interval variant, with a brief comment at the end regarding the left-length variant. The target for this example is $n = 29$. For the other parameters we let $e = 2/3$, $f = 1$, and $s = 1.1$.

Let us start with the left side where we will search for chains which reach a target between $\lceil \lceil 29^{2/3} \rceil / 1.1 \rceil = 10$ and $\lfloor \lceil 29^{2/3} \rceil \cdot 1.1 \rfloor = 11$, with pruning similar to Section 3. We find the following set of chains for the left side:

$$1, 2, 3, 5, 8, 11$$
$$1, 2, 3, 5, 7, 9, 11$$
$$1, 2, 3, 4, 7, 11$$
$$1, 2, 3, 4, 7, 10$$
$$1, 2, 3, 4, 5, 6, 11$$

These left-side chains $\ldots, b, c$ all have $6 \le b \le 9$ and $10 \le c \le 11$, giving bounds used later on combinations such as $2b + c$.

The next step is to choose a modulus $\lfloor 29^{1/3} \rfloor = 3$ and create a dictionary of the left-side chains $\ldots, b, c$ indexed by $(b \bmod 3, c \bmod 3)$. This dictionary contains the

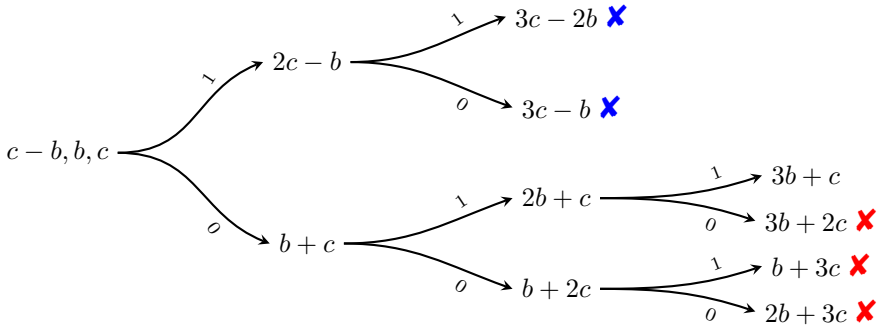

FIGURE 3. Right-side chains considered inside a particular computation for $n = 29$, noting which clearly undershoot (✘) or clearly overshoot (✘).

following:

$$(0,2) : 1,2,3,5,7,9,11; 1,2,3,4,5,6,11$$
$$(1,1) : 1,2,3,4,7,10$$
$$(1,2) : 1,2,3,4,7,11$$
$$(2,2) : 1,2,3,5,8,11$$

The goal is now to search suitable right sides, and find matching combinations of left sides and right sides. To decide how far to search within the tree of right-side chains from Figure 2, we first choose a bound on the length of the full chain. We start by considering chain length at most $\lceil 1.44 \log_2(29-1) - 2.34 \rceil = 5$, not counting the initial 2 additions for $1, 2, 3$.

The minimum length we have found on the left side is 3, so limiting the full chain length to at most 5 also limits the right side to length at most $5 - 3 = 2$. For this length limit, the beyond-guaranteed pruning from Section 4.6 eliminates the root node $c - b, b, c$ of the right-side search, since it assumes that 2 steps will gain at most a factor 2 starting from $c \leq 11$.

We thus move on to considering full chain length at most 6. This limits the right side to length at most 3. A recursive search considers the right-side chains shown in Figure 3. There are now six right-side chains that are not discarded ($c-b, b, c, 2c-b$; $c-b, b, c, b+c, b+2c$; $c-b, b, c, b+c, 2b+c, 3b+c$; and their truncations, see below). Note that this includes some chains of length 2 or less (disregarding the first three elements of the chain as usual). This is due to the fact that the pruning bounds are different since we have a longer length chain.

Each of these six right-side chains has final element $pb + qc$ for some integers p, q. For each chain, we solve the linear equation $pb + qc \equiv n \pmod 3$ to obtain a list of pairs ($b \bmod 3, c \bmod 3$), and we then check those indices in the dictionary. For example, for the chain $c - b, b, c, b + c, b + 2c$, we solve $b + 2c = 29$ modulo 3, obtaining the pairs $(0, 1)$, $(1, 2)$, and $(2, 0)$. One of these pairs, namely $(1, 2)$, appears in the dictionary, showing left-side chain $1, 2, 3, 4, 7, 11$.

Repeating this process for all right-side chains leads us to the following combinations to consider:

$c-b, b, c : 1, 2, 3, 5, 7, 9, 11; 1, 2, 3, 4, 5, 6, 11; 1, 2, 3, 4, 7, 11; 1, 2, 3, 5, 8, 11$

$c-b, b, c, b+c : 1, 2, 3, 5, 7, 9, 11; 1, 2, 3, 4, 5, 6, 11; 1, 2, 3, 4, 7, 10$

$c-b, b, c, b+c, b+2c : 1, 2, 3, 4, 7, 11$

$c-b, b, c, b+c, 2b+c : 1, 2, 3, 5, 7, 9, 11; 1, 2, 3, 4, 5, 6, 11$

$\ldots, 3b+c : 1, 2, 3, 5, 7, 9, 11; 1, 2, 3, 4, 5, 6, 11; 1, 2, 3, 4, 7, 11; 1, 2, 3, 5, 8, 11$

$c-b, b, c, 2c-b : 1, 2, 3, 5, 8, 11$

We test each of these 15 combinations (rather than 30 combinations of 5 left-side chains and 6 right-side chains). For example, on the last line, we compute $2c-b = 14 \neq 29$ from $(b, c) = (8, 11)$; on the line before, we compute $3b+c = 38 \neq 29$ from $(b, c) = (9, 11)$, we compute $3b+c = 29$ from $(b, c) = (6, 11)$, etc. Some of these combinations give $pb+qc = n$ as desired. We choose a minimum-length combination having $pb + qc = n$: specifically, $1, 2, 3, 4, 7, 11$ and $c-b, b, c, b+c, b+2c$ combine to the continued-fraction chain $1, 2, 3, 4, 7, 11, 18, 29$.

The left-length variant has a similar workflow to the left-interval variant above. The main change is that, within an outer loop incrementing lengths of the entire chain as in Section 3, the left side is accorded 2/3 of this chain, and the right side the rest. Left-side chains are put into a dictionary as above, and right-side chains are compared to the dictionary as above. The first chain found is guaranteed to be a shortest continued-fraction chain (and pruning details are chosen to maintain this guarantee). This results in the same continued-fraction chain as found in the left-interval variant for $n = 29$.

5. Results and Comparisons

The following figures compare the observed performance of various algorithms that, given a prime p, compute a chain for p. The algorithms compared here are as follows:

- "ladder": a straightforward computation of a ladder for the given prime p.
- "cfrc": Montgomery's CFRC algorithm [17, Section 5], repeated across all choices of Montgomery's auxiliary parameter ρ.
- "prac": Montgomery's PRAC algorithm [17, Section 7].
- "sibc": the algorithm from the SIBC library [1], enumerating all continued-fraction differential addition chains having at most $2+\lfloor 1.5 \log_2 p \rfloor$ additions, except for pruning chains that go beyond p.
- "ctidh": the algorithm from the CTIDH software [2], similar to "sibc" but with incremental length searching and more pruning.
- "prune": this paper's low-memory algorithm, similar to "ctidh" but with even more pruning.
- "mitm": the left-interval variant of this paper's MitM algorithm.
- "mitm2": the left-length variant of this paper's MitM algorithm.

The main features compared are (1) chain lengths and (2) the number of internal nodes visited in the algorithms. Beware that the algorithms perform different amounts of work per node; a proper run-time comparison would require replacing the Python implementation of each algorithm with an optimized low-level implementation.

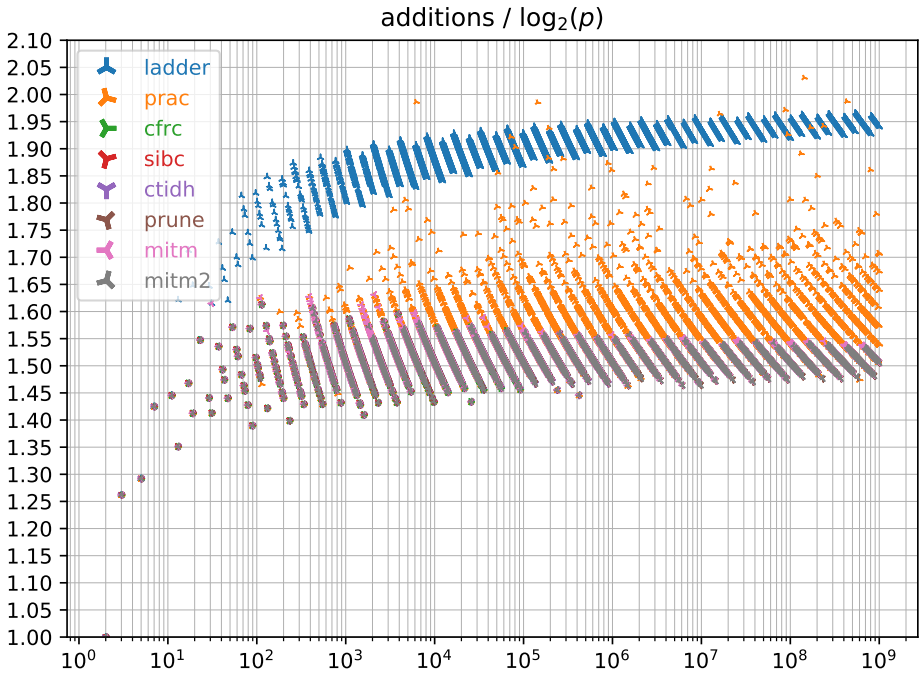

FIGURE 4. Scatterplot showing, for various small primes p, observed length of chain output by various algorithms given p as input. Horizontal axis is p. Vertical axis is additions$_p/\log_2 p$.

Each figure selects 3450 primes $p < 10^9$; the horizontal axis in each figure is p in log scale. Specifically, the primes used are as follows: start by setting $p \leftarrow 2$; as long as $p < 10^9$, find the smallest prime $q \geq p + 1 + \lfloor p/256 \rfloor$, and set $p \leftarrow q$. This covers all small primes p, but gradually shifts towards a geometric progression of larger primes p. Data for "sibc", "cfrc", and "ctidh" is collected only for $p < 10^4$, $p < 10^5$, and $p < 10^6$ respectively; these cutoffs are visible in, e.g., Figure 6 below.

The underlying data points were collected by a script `bench.py` using 11141 core-minutes of user time and 38 core-minutes of system time on a dual AMD EPYC 7742 running at 2.245GHz (no boost) running Debian 11 with Python 3.9.2. The script used 128 threads, together using about 200GB of RAM, and finished in 90 minutes of real time. The graphs were produced in 10 seconds by a script `plot.py` taking as input the output of `bench.py`. The software is available from https://cr.yp.to/2024/dacbench-20240609.tar.gz.

Figure 4 shows chain lengths. The vertical axis is the number of additions divided by $\log_2 p$, so each plotted point has the form $(p, r/\log_2 p)$ for some integer r. Note that, for each r, the curve $(p, r/\log_2 p)$ is visually a hyperbola since the horizontal axis is log scale.

The "ladder" points in the figure move up towards the asymptote 2. The long-short pattern reflects Montgomery's formula $\lfloor \log_2 p \rfloor + \lfloor \log_2(2p/3) \rfloor$ from [17, Section 2] for the ladder lengths.

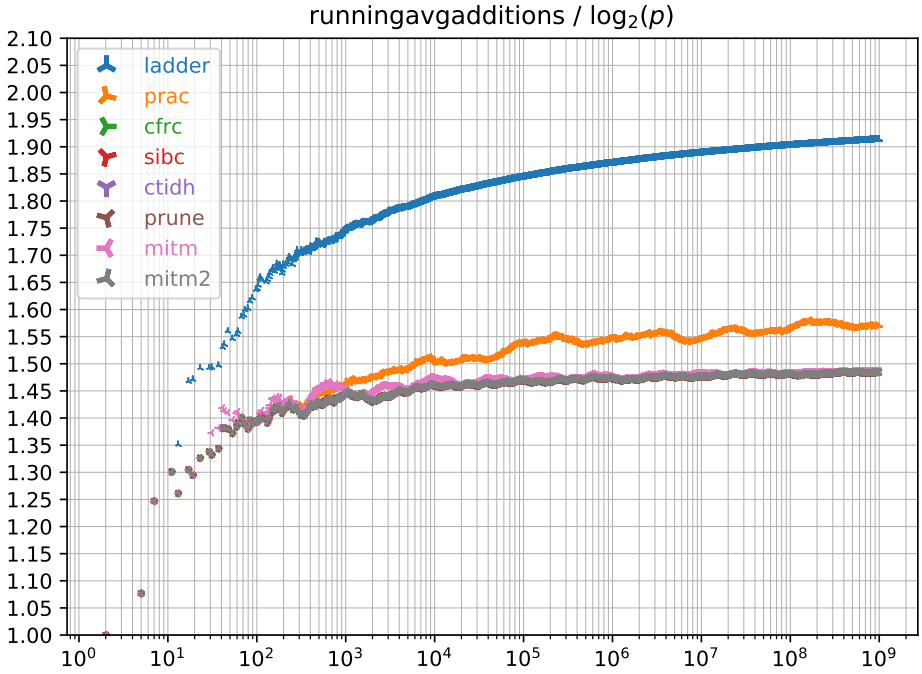

FIGURE 5. Running-average plot showing, for various small primes p, observed length of chain output by various algorithms given p as input. Horizontal axis is p. Vertical axis is $\text{runningavgadditions}_p / \log_2 p$, where $\text{runningavgadditions}_p$ is the average of additions_q over plotted primes q with $p/2 < q \le p$.

The "cfrc", "sibc", "ctidh", "prune", and "mitm2" points are always on top of each other in this figure, except for the different cutoffs on p mentioned above. The "mitm" points are occasionally higher because of how we chose parameters in the left-interval variant of the MitM algorithm. The points for all of these algorithms are frequently above 1.5 and frequently below 1.5. Recall from Section 1 that differential addition chains for primes have a lower bound, which on this graph is asymptotically the constant $1/\log_2((1+\sqrt{5})/2) \approx 1.44042$.

The "prac" points are often noticeably higher (and occasionally lower; recall that PRAC generates differential addition-subtraction chains but not necessarily continued-fraction differential addition chains). We did not experiment with Montgomery's suggestion of trying multiple ρ choices inside PRAC.

Figures 5 is a smoothed version of Figure 4, using a running average of chain lengths. Specifically, instead of plotting $(p, r_p/\log_2 p)$, Figure 5 plots $(p, s_p/\log_2 p)$ where s_p is the average of r_q over the selected primes q satisfying $p/2 < q \le p$. One easily sees in this figure that "mitm" with our parameter choices is marginally worse on average than "mitm2", but by less than 1% for large p.

Figure 6 compares the number of internal nodes visited in the algorithms. The vertical axis is $(\log_2(1+\text{nodes}_p))/\log_2 p$, so an algorithm with an essentially linear number of nodes would be asymptotically at vertical position 1, and an algorithm

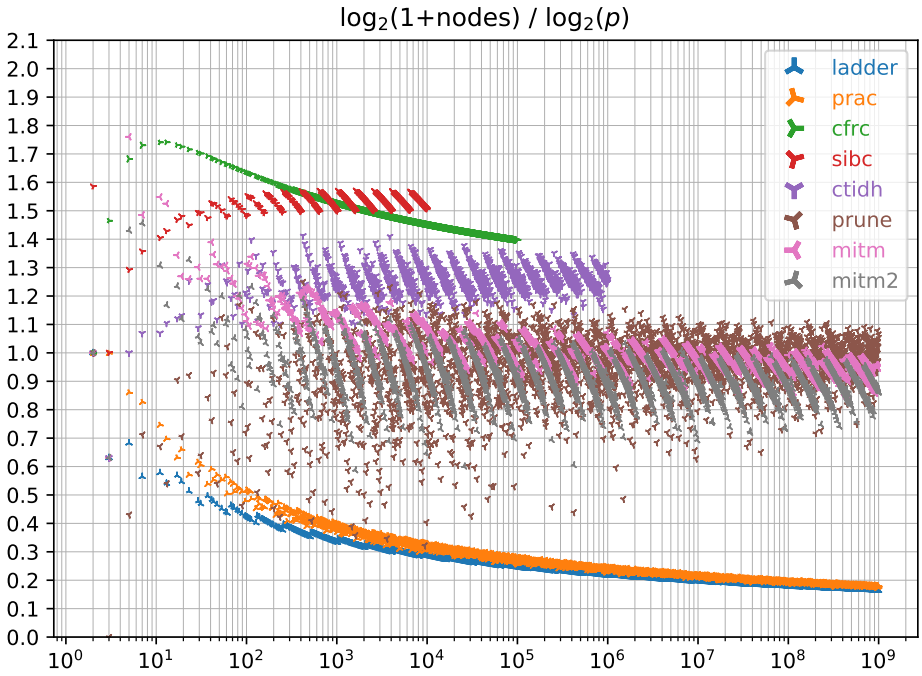

FIGURE 6. Scatterplot showing, for various small primes p, observed number of internal nodes in various algorithms given p as input. Horizontal axis is p. Vertical axis is $\log_2(1+\text{nodes}_p)$ divided by $\log_2 p$.

with an essentially quadratic number of nodes would be asymptotically at vertical position 2.

At the bottom of the figure, "ladder" and "prac" use only a logarithmic number of nodes, so they have an asymptote of 0. At the top of the figure, "cfrc" searches a linear number of choices of the auxiliary parameter, and uses $\Theta(p(\log p)^2)$ nodes, so it has an asymptote of 1. All three of these algorithms are noticeably above their asymptotes because of the logarithmic factors.

Close to the top of the figure—and at the top once p passes about 10^3—is "sibc". There are about $p^{1.5}$ continued-fraction differential addition chains of the length searched by "sibc", and most of them are searched (see Section 3.2), so "sibc" has an asymptote of 1.5. The vertical jumps in the "sibc" graph appear when $1.5\log_2 p$ crosses past an integer, doubling the number of chains searched. One can see in the figure that the rightmost "sibc" jump for $p < 10^4$ is a vertical jump by something less than 0.1; this jump is for $p \approx 2^{19/1.5} \approx 6502$, where $(\log_2 2)/\log_2 p \approx 1.5/19 \approx 0.079$. Asymptotically, these jumps converge to 0.

The performance of "ctidh" and "prune" is not as easy to analyze. The figure shows that "prune" consistently uses fewer nodes than "ctidh", which consistently uses fewer nodes than "sibc", as one would expect from the structure of the algorithms. It seems reasonable to conjecture that "prune" has *lower* asymptote 0, and more precisely that there are infinitely many primes for which "prune" visits

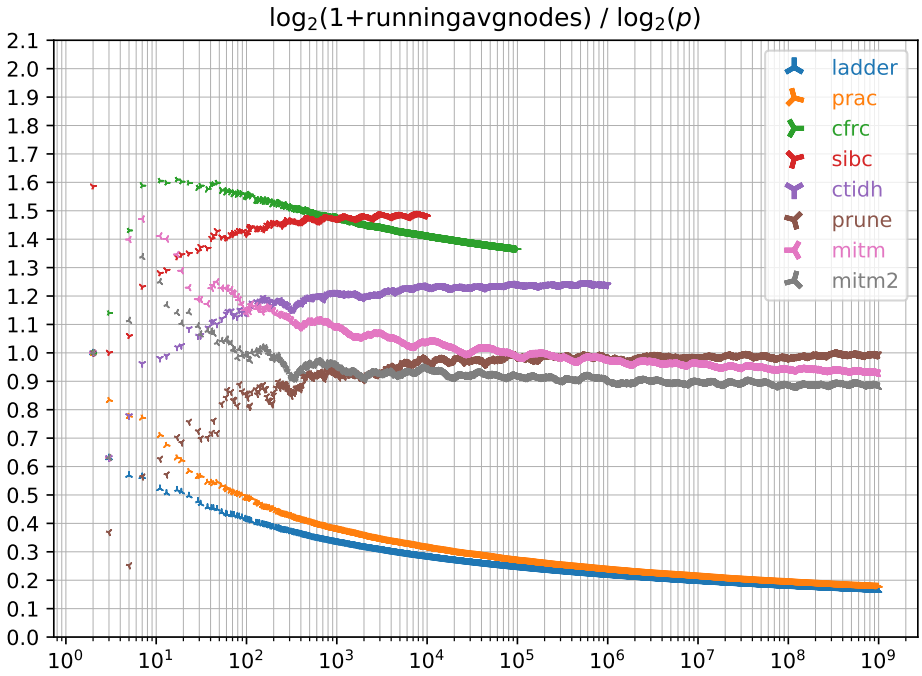

FIGURE 7. Running-average plot showing, for various small primes p, observed number of internal nodes in various algorithms given p as input. Vertical axis is $\log_2(1+\text{runningavgnodes}_p)/\log_2 p$, where runningavgnodes_p is the average of nodes_q over plotted primes q with $p/2 < q \le p$.

only a polylogarithmic number of nodes; for example, it is conjectured that there are infinitely many Fibonacci primes. On the other hand, presumably *most* inputs p have the following property: "prune" searches chains whose ends cover a considerable fraction of all integers close to p. Perhaps it is possible to prove via such considerations that "prune" has upper asymptote at least 1.

A similar argument suggests that "mitm" and "mitm2" have upper asymptote at least $2/3$. As p increases within the range covered by the figure, one can see "mitm" and "mitm2" generally decreasing, often using fewer nodes than "prune" but often using more nodes than "prune". A sanity check of the number of nanoseconds used by "prune", "mitm", and "mitm2" shows "mitm" and "mitm2" having more overhead but still outperforming "prune" more and more often as p increases. Of course, these experiments do not establish the asymptotics of these algorithms.

Finally, Figure 7 is a smoothed version of Figure 6, averaging the number of nodes in the same way that Figure 5 averages chain lengths. Within the range of the figure, "prune", "mitm", and "mitm2" use many more nodes than "ladder" and "prac" but produce shorter chains; meanwhile "prune", "mitm", and "mitm2" use fewer nodes than "cfrc", "sibc", and "ctidh", while producing practically the same chain length, or, in the case of "prune" and "mitm2", exactly the same chain length.

References

[1] Gora Adj, Jesús-Javier Chi-Domínguez, and Francisco Rodríguez-Henríquez. SIBC Python library. https://github.com/JJChiDguez/sibc/, 2021. 4, 8, 17

[2] Gustavo Banegas, Daniel J. Bernstein, Fabio Campos, Tung Chou, Tanja Lange, Michael Meyer, Benjamin Smith, and Jana Sotáková. CTIDH: faster constant-time CSIDH. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):351–387, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9069. doi:10.46586/tches.v2021.i4.351-387. 3, 4, 5, 8, 10, 17

[3] Daniel J. Bernstein. Differential addition chains, 2006. URL: https://cr.yp.to/papers.html#diffchain. 5

[4] Daniel J. Bernstein and Tanja Lange. Explicit-Formulas Database. Accessed 8 June 2024. URL: https://hyperelliptic.org/EFD. 3, 6

[5] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50, Kuching, Malaysia, December 2–6, 2007. Springer, Heidelberg, Germany. doi:10.1007/978-3-540-76900-2_3. 2

[6] Daniel Bleichenbacher and Achim Flammenkamp. An efficient algorithm for computing shortest addition chains, 1997. URL: https://wwwhomes.uni-bielefeld.de/achim/addition_chain_bibliography.html. 5

[7] Daniel Cervantes-Vázquez, Mathilde Chenu, Jesús-Javier Chi-Domínguez, Luca De Feo, Francisco Rodríguez-Henríquez, and Benjamin Smith. Stronger and faster side-channel protections for CSIDH. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology - LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America*, volume 11774 of *Lecture Notes in Computer Science*, pages 173–193, Santiago, Chile, October 2–4, 2019. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-30530-7_9. 4, 5

[8] Yang-Ho Chin and Yueh-Hsia Tsai. Algorithms for finding the shortest addition chain. In *Proceedings of the 1985 National Computer Symposium of the Republic of China*, pages 1398–1414, 1986. 5

[9] Neill Michael Clift. Calculating optimal addition chains. *Computing*, 91(3):265–284, 2011. doi:10.1007/S00607-010-0118-8. 5

[10] Pierrick Dartois, Antonin Leroux, Damien Robert, and Benjamin Wesolowski. SQISignHD: New dimensions in cryptography. 2023. URL: https://eprint.iacr.org/2023/436. 3

[11] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 64–93, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-64837-4_3. 4

[12] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany. doi:10.1007/3-540-44647-8_11. 5

[13] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n \log n)$. *Annals of Mathematics. Second Series*, 193:563–617, 2021. 3

[14] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards curves revisited. In Josef Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343, Melbourne, Australia, December 7–11, 2008. Springer, Heidelberg, Germany. doi:10.1007/978-3-540-89255-7_20. 2

[15] Donald E. Knuth. ACHAIN0, ACHAIN1, ACHAIN2, ACHAIN3, ACHAIN4, and ACHAINALL, 2005. URL: https://www-cs-faculty.stanford.edu/~knuth/programs.html. 5

[16] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. 1

[17] Peter L. Montgomery. Evaluating recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via Lucas chains, 1992. URL: https://cr.yp.to/bib/1992/montgomery-lucas.pdf. 2, 3, 4, 5, 8, 17, 18

[18] Arnold Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1(2):139–144, 1971. 3

[19] Martijn Stam. *Speeding up subgroup cryptosystems*. PhD thesis, Technische Universiteit Eindhoven, 2003. URL: https://pure.tue.nl/ws/portalfiles/portal/2122664/200311829.pdf. 5

[20] Edward G. Thurber. Efficient generation of minimal length addition chains. *SIAM J. Comput.*, 28(4):1247–1263, 1999. doi:10.1137/S0097539795295663. 5

[21] Edward G. Thurber and Neill Michael Clift. Addition chains, vector chains, and efficient computation. *Discret. Math.*, 344(2):112200, 2021. doi:10.1016/J.DISC.2020.112200. 5

[22] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, January 1999. doi:10.1007/PL00003816. 5

[23] Andrew C. Yao and Donald E. Knuth. Analysis of the subtractive algorithm for greatest common divisors. *Proceedings of the National Academy of Sciences*, 72(12):4720–4722, 1975. URL: https://www.pnas.org/doi/pdf/10.1073/pnas.72.12.4720. 3

Daniel J. Bernstein, University of Illinois at Chicago, USA, and Academia Sinica, Taiwan
Email address: djb@cr.yp.to

Jolijn Cottaar, Eindhoven University of Technology, Netherlands
Email address: jolijncottaar@gmail.com

Tanja Lange, Eindhoven University of Technology, Netherlands, and Academia Sinica, Taiwan
Email address: tanja@hyperelliptic.org