Cryptology ePrint Archive, Report 2007/470,
http://eprint.iacr.org/

Faster Multi-Exponentiation through Caching: Accelerating (EC)DSA Signature Verification

Bodo Möller and Andy Rupp

Horst Görtz Institute for IT Security, Ruhr-Universität Bochum
bmoeller@acm.org, arupp@crypto.rub.de

Abstract. We consider the task of computing power products $\prod_{1 \le i \le k} g_i^{e_i}$ ("multi-exponentiation") where base elements $g_2, \ldots, g_k$ are fixed while g_1 is variable between multi-exponentiations but may repeat, and where the exponents are bounded (e.g., in a finite group). We present a new technique that entails two different ways of computing such a result. The first way applies to the first occurrence of any g_1 where, besides obtaining the actual result, we create a cache entry based on g_1, investing very little memory or time overhead. The second way applies to any multi-exponentiation once such a cache entry exists for the g_1 in question: the cache entry provides for a significant speed-up. Our technique is useful for ECDSA or DSA signature verification with common domain parameters and recurring signers.

Keywords: Efficient implementation, elliptic curve cryptography, ECDSA verification, exponentiation, DSA verification

1 Introduction

Consider a scenario where we repeatedly have to verify ECDSA signatures [1], trying to keep the computational delay small for each verification. A time-consuming step in ECDSA signature verification is computing a linear combination $u_1G + u_2Q$ of elliptic curve points G and Q, where G is specified by domain parameters and where Q constitutes the signer's public key, with integers u_1 and u_2 in the interval $\big(0, \mathrm{ord}(G) - 1\}\big)$ both depending on the specific signature. The same group with the same point G will typically be shared by many signers since elliptic curve domain parameters are often taken from (intersecting) standards such as [19, Appendix 6], [1, Annex J], and [6] (with domain parameter specifications `NIST P-192` aka `prime192v1` aka `secp192r1` and `NIST P-256` aka `prime256v1` aka `secp256r1` common to all three of these). Also, we usually can expect some signers and thus their Q values to recur. Consider public-key infrastructures:

– A verifying party will encounter end-entity certificates signed by possibly very many different intermediate certification authorities. When a new certification authority appears for the first time, the verifying party does not yet know how popular this particular certification authority is, i.e. if it has signed many or just very few end-entity certificates.

– The same applies to signatures on documents, such as the ECDSA signatures stored on the new digital "e-passports". When verifying a passport for airport immigration procedures, then quite possibly the next passenger in line may be using a passport signed by the same agency. On the other hand, the current passenger could be the only one from his particular country.

Thus, for a given G, we have to compute linear combinations $u_1G + u_2Q$ where Q sometimes is "new" and sometimes is "old" (has been seen before); but when a new Q appears for the first time, we generally do not know if it will appear again later on.

There are well-known techniques to compute $u_1G + u_2Q$ much faster than by computing both u_1G and u_2Q individually, and this can be done yet faster if both G and Q are fixed and a one-time precomputation depending on these points has been done. Performing such precomputation whenever a "new" Q shows up may pay out if Q turns out to repeat, so that G and Q are fixed for a number of linear combinations. However, precomputation depending on Q is an investment of resources that would be lost if this particular Q does in fact not repeat.

We present a new technique that nearly avoids this drawback, provided that space for permanently fixed precomputation depending on G only is not severely limited. The first occurrence of some point Q in a computation $u_1G + u_2Q$ will incur very little penalty in terms of memory or time, and yet will leave behind useful precomputed data that can be cached to speed up subsequent linear combination computations involving the same G and Q.

The ECDSA scenario best illustrates the practical use of our technique[1], which we describe in more general form. It applies to any abelian group or (more generally) abelian semigroup with an identity element, henceforth written multiplicatively so that what we just described as linear combinations now turns into power products. Computing power products sometimes is called *multi-exponentiation* since it is a generalization of computing powers (exponentiation). The computational task that we will consider is computing power products of the form

$$\prod_{1 \le i \le k} g_i^{e_i}$$

where base elements $g_2, \ldots, g_k$ are fixed once and for all, whereas g_1 is vari-

[1] Another approach to speed up ECDSA signature verification is due to Antipa et al. [2, 24]. It works best for a slightly modified variant of the original signature scheme, dubbed ECDSA*, but under appropriate circumstances, it can be useful for the verification of standard ECDSA signatures. Where it makes sense to use the technique from [2, 24], our technique may be preferable depending on the expected proportion of "old" to "new" Q values. In fact, we can get some of the benefit of [2, 24] for any "new" Q and all of the benefit of our technique for any "old" Q by using a combination of both techniques in the case of a "new" Q, using specific imbalanced-scalar parameterizations within [2, 24]. We omit further details on this.

able between multi-exponentiations and *may* repeat, while all exponents e_i are assumed to be ephemeral. We will assume that the exponents are positive and at most ℓ bits long. (An appropriate value of ℓ is usually implied by the group order. A negative exponent for a group can be handled through inversion of the base element, or by reduction of the exponent modulo o where o is some multiple of the order of the base element, such as the group order.) For our technique to work as intended, we also assume that the exponent to variable base g_1 is not pathologically short (i.e., its length not just a fraction of ℓ); rare statistical outliers are no problem. Besides ECDSA signature verification, this setting also covers DSA signature verification [19]; however, it only applies when using common domain parameters, which is customary for ECDSA but not necessarily for DSA.

We assume that at least read-only memory is not severely limited, so that precomputation depending on $g_2, \ldots, g_k$ can be permanently stored. We also assume that some memory is available for caching at least one group element with an integer. Such data will be put into cache memory when performing a multi-exponentiation $\prod_{1 \le i \le k} g_i^{e_i}$ involving a "new" g_1 (i.e., one for which no cache entry currently exists), and can be used to speed up any subsequent multi-exponentiation repeating the same g_1 as long as the cache entry is kept. While the method is easier to describe assuming that dedicated cache memory is available, Appendix C will show that the technique can be quite useful even if this is not the case and a portion of fast read/write memory has to be sacrificed instead: In a specific example scenario where read/write memory is very scarce (which is typical of smart cards and other embedded devices), we have a 10 % average speed advantage. The technique also is useful for devices without such constraints.

Like many approaches for exponentiation using precomputation (such as the Lim/Lee approach [12]), our technique has roots that can be traced back to Pippenger [21, 22]; see also [4]. The novelty here in this regard is that for cached precomputation, we do not store powers of the form $g_1^{2^n}$ as by [21], which would impose some computational overhead while computing $\prod_{1 \le i \le k} g_i^{e_i}$ when g_1 is new. Instead, we store other powers of g_1 that happen to come up without effort if we arrange the computation suitably.

Section 2 describes preliminaries for our technique: interleaved multi-exponentiation, and radix-2 exponent splitting. Then, Section 3 presents the novel multi-exponentiation technique, which relies on caching certain intermediate results that can be obtained by investing very little additional read/write memory or time, allowing us to speed up later multi-exponentiations if an appropriate cache entry is available. Section 4 gives example performance figures for the new technique in certain use scenarios. Appendix A provides a comprehensive example to illustrate the technique. Appendix B discusses some implementation aspects. Finally, Appendix C considers a particular scenario to demonstrate the performance gain that can be obtained by using the new technique.

2 Multi-Exponentiation

We show known techniques that we later will use and combine in a novel way. Section 2.1 describes *interleaved multi-exponentiation,* an approach for computing power products. It also briefly describes some properties of radix-2 exponent representations that can be used in interleaved multi-exponentiation. Section 2.2 describes the technique of *radix*-2 *exponent splitting,* which can be used to obtain shorter exponents by converting exponentiation tasks into multi-exponentiation tasks, or converting k-fold multi-exponentiation tasks into k'-fold multi-exponentiation tasks with $k' > k$. Radix-2 exponent splitting is a useful technique for fixed bases (namely, for exponentiation or multi-exponentiation with precomputation that can be done in advance).

2.1 Interleaved Multi-Exponentiation

We build on the straightforward multi-exponentiation strategy that has been called *interleaving* in [14], which generalizes well-known methods for single exponentiations such as the (left-to-right) binary or sliding window methods. Assume that radix-2 representations

$$e_i = \sum_{0 \le j \le \ell} b_{i,j} \cdot 2^j, \qquad b_{i,j} \in B_i,$$

of all exponents are given where each B_i is a set of integers. We write $e_i = (b_{i,\ell}, b_{i,\ell-1}, \ldots, b_{i,1}, b_{i,0})_2$ and call the $b_{i,j}$ *digits* and B_i a *digit set.* We require that every g_i^b for $b \in B_i \setminus \{0\}$ be available from precomputed data. Note that in a group where inversion is particularly easy (such as those used in elliptic curve cryptography where an inversion requires just obtaining an additive inverse in the underlying field or performing a field addition), obtaining g_i^{-b} from precomputed data is easy if g_i^b has been stored; so both b and $-b$ can be included in set B_i if the single element g_i^b has been stored in a precomputed table of powers. In this setting, interleaved multi-exponentiation computes the power product as follows.

$A \leftarrow 1_G$ {Start with identity element}
for $j = \ell$ down to 0 **do**
  $A \leftarrow A^2$
  for $i = 1$ to k **do**
    if $b_{i,j} \neq 0$ **then**
      $A \leftarrow A \cdot g_i^{b_{i,j}}$ {Multiply by [inverse of] precomputed element}
return A

This is a left-to-right technique in that it proceeds from the most significant digits ("left") down to the least significant digits ("right").

Digit sets usually are quite small, typically of the form $B^{\pm}(m) = \{\pm 1, \pm 3, \pm 5, \pm 7, \ldots, \pm m, 0\}$ for groups where inversion is easy, or $B(m) = \{1, 3, 5, 7, \ldots, m, 0\}$ for semigroups in general. Here parameter m is an odd integer, often but not necessarily of the form $1, (11)_2, (111)_2, \ldots, (11\ldots11)_2$, i.e.,

$2^w - 1$, $w \geq 1$ an integer. This special form applies to the sliding window technique (cf. [9]) and to various variants of it that employ signed-digit representations of exponents, such as those introduced in [13] using a right-to-left conversion from binary to signed-digit representation and in [17, 3, 20] using left-to-right conversions. The general case with an arbitrary odd m was introduced as *fractional window representations* in [15], with left-to-right conversions for the signed-digit case suggested in [11, 23, 16]. Different digits sets can be used for different exponents, so we have $B_i = B(m_i)$ or $B_i = B^{\pm}(m_i)$ with per-exponent parameters m_i when employing such representations.

The *length* of a representation is the number of digits that remain if we drop any leading zeros (so the length of $(b_\ell, b_{\ell-1}, \ldots, b_1, b_0)_2$ is $\ell + 1$ if $b_\ell \neq 0$). Maximum length $l+1$ is sufficient to represent any l-bit integer e ($2^{l-1} \leq e < 2^l$) in any of the representations mentioned above [18] (length l is sufficient for any of the unsigned-digit representations), and the minimum length with these representations is $l + 1 - \lceil \log_2 m \rceil$. Thus, the maximum outer loop index ℓ in the algorithm as shown above is sufficient for integers up to ℓ bits.

The *weight* of a representation is the number of digits that are non-zero. The conversion techniques mentioned above are known to achieve, for any integer e, the minimum weight possible given the respective digit set [18, 16]. For unsigned and signed fractional window representations using digit set $\{1, 3, 5, \ldots, m, 0\}$ or $\{\pm 1, \pm 3, \pm 5, \ldots, \pm m, 0\}$, the average weight for random integers up to ℓ bits is slightly above

$$\frac{\ell}{1 + w(m) + \dfrac{m + 1 - 2^{w(m)}}{2^{w(m)}}} \quad \text{and} \quad \frac{\ell}{2 + w(m) + \dfrac{m + 1 - 2^{w(m)}}{2^{w(m)}}},$$

respectively, where

$$w(m) = \lfloor \log_2(m + 1) \rfloor;$$

the average *density* (weight divided by ℓ) converges to according estimates as ℓ goes to ∞ (see [16]). For the special case $m = 2^w - 1$ (i.e., the sliding window technique and its non-fractional signed-digit counterparts), such that $w(m) = w$, the above is simply $\ell/(1 + w)$ and $\ell/(2 + w)$, respectively.

Observe that in the interleaved multi-exponentiation algorithm as shown above, (semi-)group operations need not actually be performed until after the first multiplication of A by a precomputed element or its inverse, since $A = 1_G$ holds up to this point. This means that the initial squarings of 1_G can be skipped, and the first operation $A \leftarrow A \cdot g_i^{b_{i,j}}$ amounts to an assignment $A \leftarrow g_i^{b_{i,j}}$.

To estimate the time required to perform an interleaved multi-exponentiation, we thus need the maximum length of the representations of the e_i to determine the number of squarings, and the weight of the representation of each e_i to determine the number of other multiplications by elements available from precomputed data. (The maximum length is one more than the number of squarings, and the sum of the weights is one more than the number of other multiplications.) This is not counting any group inversions, since we would only use these in the algorithm if inversion is easy. In addition to this,

we have to look at the time needed for precomputation. If g_i is a fixed base, we can assume that the required powers g_i^b have been precomputed in advance (and possibly built into ROM) and thus do not enter the time estimate. However, if g_i is not fixed, some effort goes into precomputing these powers: from g_i, the powers $g_i^3, \ldots, g_i^{m_i}$ can be computed using one squaring (to obtain g_i^2 as an intermediate value) and $\frac{m_i-1}{2}$ other multiplications.

(Note that the minimum-weight property of a conversion technique does not mean that it always provides the best radix-2 representation possible given the particular digit set. As discussed in [15, Section 5.1] and [16, Section 4], *modified signed fractional window representations* can sometimes reduce length without increasing weight. In certain situations, it may even be of advantage to accept a slight increase of weight for the sake of length reduction if saved squarings [due to length reduction] outweigh the additional multiplications [due to weight increase]. To pursue this approach, we can generalize the concept of radix-2 representations: e.g., $(100000)_2 = 2^5$ could be converted into $3 \cdot 2^2 + 5 \cdot 2^2$, which is not a proper radix-2 representation but might be written as $\binom{3}{5}00\Big)_2$ using a "double digit" of weight 2. Details are out of the scope of the present paper; we just mention this as a reminder that minimum-weight radix-2 representations can sometimes be improved by applying appropriate substitution rules.)

2.2 Radix-2 Exponent Splitting

We have seen that the length of exponent representations is important to efficiency since it determines the number of squarings needed for interleaved multi-exponentiation. For an exponent e, this length is around $\log_2 e$ with any of the representations mentioned in Section 2.1 as long as parameter m is reasonably small. *Radix-2 exponent splitting,* shown in the following, is a simple but effective idea (underlying [5] and made explicit in [8]) to get better performance if all bases are fixed.

For exponentiations g^e with exponent representations $e = (b_\ell, \ldots, b_0)_2$ of maximum length $\ell + 1$, we can decide to split each such representation into some number s of shorter exponent representations. To wit, let $\ell + 1 = L_1 + \cdots + L_s$ with positive integers $L_i \approx \frac{\ell+1}{s}$, and then let $e_1 = (b_{L_1-1}, \ldots, b_0)_2$, $e_2 = (b_{L_1+L_2-1}, \ldots, b_{L_1})_2$, and so on:

$$\begin{aligned} e = (b_\ell, \ldots, b_0)_2 = (&\underbrace{b_{L_1+\cdots+L_s-1}, \ldots, b_{L_1+\cdots+L_{s-1}}}_{e_s}, \\ &\underbrace{b_{L_1+\cdots+L_{s-1}-1}, \ldots, b_{L_1+\cdots+L_{s-2}}}_{e_{s-1}}, \ \ldots, \\ &\underbrace{b_{L_1+L_2-1}, \ldots, b_{L_1}}_{e_2}, \ \underbrace{b_{L_1-1}, \ldots, b_0}_{e_1})_2 \end{aligned}$$

Then from $e = \sum_{1 \le i \le s} e_i \cdot 2^{L_1+\cdots+L_{i-1}}$ it follows that

$$g^e = g^{e_1} \cdot \left(g^{2^{L_1}}\right)^{e_2} \cdot \ \cdots \ \cdot \left(g^{2^{L_1+\cdots+L_{s-2}}}\right)^{e_{s-1}} \cdot \left(g^{2^{L_1+\cdots+L_{s-1}}}\right)^{e_s},$$

and thus by defining $g_i = g^{2^{\sum_{1 \le I < i} L_I}}$ we have transformed the task of computing g^e into the s-fold multi-exponentiation $\prod_{1 \le i \le s} g_i^{e_i}$. There is no need to actually evaluate the e_i as integers here since we already have appropriate representations of them—namely, portions of the original representation as shown.

Thanks to exponent splitting, the maximum length of exponent representations can go down from $\ell + 1$ to $\lceil \frac{\ell+1}{s} \rceil$ if the L_i are chosen accordingly. If g is fixed (and the parameters L_i are constant), then so are the g_i as defined here. Thus, the powers g_i^b needed by the interleaved multi-exponentiation algorithm in Section 2.1 can be precomputed in advance. So using additional memory for precomputed data (possibly ROM) allows us to save time in each exponentiation.

So far, we have looked at radix-2 exponent splitting applied to exponentiation, not to multi-exponentiation: each single exponentiation is converted into a multi-exponentiation. Radix-2 exponent splitting can just as well be applied for any fixed base in multi-exponentiation tasks, converting a k-fold multi-exponentiation into some k'-fold multi-exponentiation, $k' > k$. However, since the exponent splitting technique needs additional precomputed data (the powers $g_i = g^{2^{\sum_{1 \le I < i} L_I}}$ of base g), it cannot be used to advantage for bases that are not fixed. Thus, if there is any base that is not fixed (as in the case of DSA and ECDSA signature verification), long exponent representations may remain, and radix-2 exponent splitting hence will typically provide essentially no speed advantage in this situation.

3 Faster Multi-Exponentiation by Caching Intermediate Results

This section describes a novel technique for computing power products $\prod_{1 \le i \le k} g_i^{e_i}$ assuming that $g_2, \ldots, g_k$ are fixed base elements, while g_1 is a variable base element whose values may recur. The technique is based on interleaved multi-exponentiation and on exponent splitting, but adds new features. It consists of two different multi-exponentiation algorithms. The first algorithm, described below in Section 3.1, is employed whenever a "new" g_1 value appears. This algorithm not only computes the multi-exponentiation result, but also outputs certain intermediate results, intended to be cached for later use. The second algorithm, described below in Section 3.2, can be employed whenever an "old" g_1 value appears, namely one for which a cache entry already exists. This algorithm then exploits the cache entry created by the first algorithm to compute the new multi-exponentiation result faster.

For both algorithms, we assume that parameters for radix-2 exponent splitting have been fixed, i.e. we have constant integers s and $L_1, \ldots, L_s$ as described in Section 2.2, used identically for all bases $g_2, \ldots, g_k$. We demand that $L_1 + 1 \ge \max_{1 \le i \le s} L_i$. For these bases, we furthermore assume that digit sets for exponent representations have been fixed (see Section 2.1), and that there is a fixed length limit $\ell + 1$ for exponent representations. (This is enough for exponents up to ℓ bits, using any of the representations mentioned in Section 2.1.) We also require that powers of $g_2, \ldots, g_k$ as required for radix-2 exponent splitting

using the given digit sets and exponent splitting parameters are precomputed in advance. These are constant elements, so they may be stored in ROM. Due to our assumption that at least read-only memory is not severely limited, it should be possible to store quite a number of such precomputed elements, allowing us to use reasonably large digit sets in the representations of exponents $e_2, \ldots, e_k$ that will undergo radix-2 exponent splitting.

Of course, since cache entries take up read/write memory, they eventually may have to be expired as new g_1 values occur. Once the cache entry for a certain g_1 has been deleted, this particular value again will have to be considered "new" if it occurs once more later. In extreme cases, the cache might provide space just for a single cache entry. Then, depending on the caching strategy implemented, g_1 might be recognized as "old" only if two immediately consecutive multi-exponentiations involve the same g_1 value, since any new value might lead to an instant cache eviction to make space for a new entry. However, it would also be possible to keep the existing cache entry for a while even if new g_1 values appear, meaning that any cacheable data created for such a new g_1 value would have to be discarded for lack of space. Which caching strategy is to be preferred depends very much on the statistical properties of the application scenario.

3.1 Multi-Exponentiation for a New Base g_1

If no cache entry based on g_1 is available, $\prod_{1 \le i \le k} g_i^{e_i}$ should be computed as follows. As in Section 2.1, we assume that the exponents are given in representations $e_i = \sum_{0 \le j \le \ell} b_{i,j} \cdot 2^j$, $b_{i,j} \in B_i$.

First, apply radix-2 exponent splitting (Section 2.2) to the representations of exponents e_2 through e_k such that all of the resulting exponent representations observe maximum length $L = \max_{1 \le i \le s} L_i$ ($\approx \frac{\ell+1}{s}$). This transforms the k-fold multi-exponentiation task into a multi-exponentiation task with more bases, where the exponent to g_1 appears unchanged but all other exponent representations have been split into parts no longer than L digits. The list of bases has expanded from $(g_1, g_2, \ldots, g_k)$ into

$$\big(g_1, \quad g_2, g_2^{2^{L_1}}, \ldots, g_2^{2^{L_1+\cdots+L_{s-1}}}, \quad \ldots, \quad g_k, g_k^{2^{L_1}}, \ldots, g_k^{2^{L_1+\cdots+L_{s-1}}}\big);$$

we will assume that g_1 keeps its index ($i = 1$). Now apply the interleaved multi-exponentiation algorithm from Section 2.1 to this new $\big(1 + (k-1)s\big)$-fold power product. (Remember that appropriate precomputed powers of the bases except g_1 are assumed to be available e.g. from ROM.) This will generate the desired result, $\prod_{1 \le i \le k} g_i^{e_i}$. Additionally, it will generate certain intermediate values that turn out to be very useful.

Observe that no loop iteration before $j = L_1$ may involve non-zero exponent digits for any base other than g_1 (since we have $L_1 + 1 \ge L_i$ for any of the exponent splitting parameters $L_2, \ldots, L_s$). In other words, before this round, A has never been multiplied with a power of a base other than g_1 . In particular, we have $A = g_1^{(b_{1,\ell},\ldots,b_{1,L_1})_2}$ just after the inner loop iteration for $j = L_1, i = 1$

(and still after the outer loop iteration for $j = L_1$ if $L_1 \geq \max_i L_i$). From this and earlier loop iterations, we can obtain the following $s-1$ intermediate values:

$$\begin{aligned} j = L_1 + \cdots + L_{s-1} \;&\Rightarrow\; A = g_1^{(b_{1,\ell},\ldots,b_{1,L_1+\cdots+L_{s-1}})_2} \\ &\cdots \qquad \cdots \\ j = L_1 + L_2 \;&\Rightarrow\; A = g_1^{(b_{1,\ell},\ldots,b_{1,L_1+L_2})_2} \\ j = L_1, i = 1 \;&\Rightarrow\; A = g_1^{(b_{1,\ell},\ldots,b_{1,L_1})_2} \end{aligned}$$

Thus, we can output the following data to be cached—a *cache entry* comprising g_1 itself (as index to the cache) and $s-1$ pairs, each consisting of an integer and the corresponding power of g_1:

$$\begin{gathered} \Big(g_1,\ \Big((b_{1,\ell},\ldots,b_{1,L_1})_2,\ g_1^{(b_{1,\ell},\ldots,b_{1,L_1})_2}\Big), \\ \Big((b_{1,\ell},\ldots,b_{1,L_1+L_2})_2,\ g_1^{(b_{1,\ell},\ldots,b_{1,L_1+L_2})_2}\Big), \\ \ldots, \\ \Big((b_{1,\ell},\ldots,b_{1,L_1+\cdots+L_{s-1}})_2,\ g_1^{(b_{1,\ell},\ldots,b_{1,L_1+\cdots+L_{s-1}})_2}\Big)\Big) \end{gathered}$$

Note that when writing this to cache, the integers may be evaluated as such—there is no need to store the specific radix-2 representations. (However, since all of these integers are derived from e_1 following a fixed rule, it is clear that at most ℓ bits are sufficient to store complete information on all of them, should memory efficiency be an utmost concern.) With any of the representations mentioned in Section 2.1, these partial integers are guaranteed to be non-negative, with

$$(b_{1,\ell},\ldots,b_{1,L_1})_2 \;\geq\; \ldots \;\geq\; (b_{1,\ell},\ldots,b_{1,L_1+\cdots+L_{s-1}})_2 \;\geq\; 0.$$

Furthermore, if e_1 is uniformly random from some set $(0,\ldots,q)$ of integers where q is an ℓ-bit integer, then (unless L_s is very small) all of these integers will actually be positive with high probability (and will be reasonably close to $2^{\ell-L_1},\ldots,2^{\ell-L_1-\cdots-L_{s-1}}$, respectively; i.e., since $\ell = \sum_{1\leq i\leq s} L_i$, to $2^{L_2+\cdots+L_s},\ldots,2^{L_s}$).

Depending on the assumed distribution of e_1, it may be a good idea to skip writing a cache entry if it ever turns out that $(b_{1,\ell},\ldots,b_{1,L_1+\cdots+L_{s-1}})_2 = 0$. In any case, writing a cache entry should be skipped if all of the integers in it would be zero (and thus the corresponding powers of g_1 trivial).

3.2 Multi-Exponentiation for an Old Base g_1

If a cache entry based on g_1 is available (created as described in Section 3.1), then $\prod_{1\leq i\leq k} g_i^{e_i}$ may be computed as follows.

First, parse the cache entry as

$$\big(g_1,\ (\lambda_1, G_1),\ \ldots,\ (\lambda_{s-1}, G_{s-1})\big).$$

Here we have $G_i = g_1^{\lambda_i}$ for $1 \leq i \leq s-1$, and if one of the exponent representations mentioned in Section 2.1 was used while creating the cache entry as specified in Section 3.1, we have $\lambda_1 \geq \ldots \geq \lambda_{s-1} \geq 0$. Now split e_1 into integers E_i $(1 \leq i \leq s)$ as follows:

- let $d_0 = e_1$;
- for $1 \leq i \leq s-1$, let $E_i = \left\lfloor \frac{d_{i-1}}{\lambda_i} \right\rfloor$ and $d_i = d_{i-1} - E_i\lambda_i$;
- and finally, let $E_s = d_{s-1}$.

In the exceptional case that $\lambda_i = 0$, $E_i = 0$ should be substituted for $\left\lfloor \frac{d_{i-1}}{\lambda_i} \right\rfloor$. By this construction, we have $e_1 = E_1\lambda_1 + \cdots + E_{s-1}\lambda_{s-1} + E_s$. It follows that

$$g_1^{e_1} = G_1^{E_1} \cdot \ \cdots \ \cdot G_{s-1}^{E_{s-1}} \cdot g_1^{E_s},$$

and thus we have transformed the power $g_1^{e_1}$ into a power product using new exponents E_i. This step is similar to radix-2 exponent splitting; we call it *modular exponent splitting.* Suitable digit sets for radix-2 representations of each of the new exponents can be chosen depending on how much read/write memory is available for storing powers of the bases $G_1, \ldots, G_{s-1}$ and g_1 (cf. Section 2.1).

For the exponents to the fixed bases $g_2, \ldots, g_k$, we again (exactly as in Section 3.1) assume that these are given in representations $e_i = \sum_{0 \leq j \leq \ell} b_{i,j} \cdot 2^j$, $b_{i,j} \in B_i$. We apply radix-2 exponent splitting to these, giving us exponent representations of maximum length L.

In total, by applying both modular exponent splitting and radix-2 exponent splitting, we have converted the k-fold multi-exponentiation into a ks-fold multi-exponentiation. The maximum length of exponent representations here may exceed L since we do not have strict guarantees regarding the E_i. However, under the assumptions regarding the distribution of e_1 stated in Section 3.1, the maximum length will remain around L with high probability.

This completes the description of our new technique. For an illustrative example we refer the reader to Appendix A.

4 Performance

Our multi-exponentiation technique can be used under many different parameterizations—the number of bases may vary; the length of exponents may vary; the amount of memory available for fixed precomputation (such as ROM) may vary; the amount of memory available for cache entries (such as slow read/write memory) may vary; the amount of memory available for variable precomputed elements needed by the interleaved exponentiation algorithm may vary; and under any of these parameterizations, we have to decide on parameters s and $L_1, \ldots, L_s$ for exponent splitting (s-fold exponent splitting with exponent segment lengths L_i), and we have to decide on digit sets and representation conversion techniques for the exponents to the fixed bases $g_2, \ldots, g_k$ on the one hand, and for *any* of the s partial exponents created from e_1 when the algorithm from Section 3.2 uses a cache entry on the other hand. This encompasses a large variety of different settings.

In the present section, we will look at a specific range of rather simple use scenarios for our new technique to assess its performance. For a comparison with other approaches, see also Appendix C.

Here, let us assume that we want to implement the multi-exponentiation technique in an environment where only a very limited amount of fast read/write memory is available but where we have some slower memory suitable for the cache, and where we have plenty of read-only memory for permanently fixed precomputed elements. As powers of g_1 are frequently needed in the course of the algorithm, this is what we will use such fast memory for. As particular examples, let us consider the cases where we have such fast memory space to store 4, 8, 16 or 32 group elements, and let ℓ be 160, 192 or 256, which are practical values for ECDSA. Note that restricting the space for storing powers of a base also limits the number of different digit values that we can use in exponent representations for the interleaved multi-exponentiation algorithm. We have implemented our new multi-exponentiation strategy and counted certain group operations under these prerequisites for different values of the splitting parameter s, always using reasonable $L_i \approx \frac{\ell+1}{s}$ and a left-to-right signed fractional window representation using appropriate digit sets $B^{\pm}(m) = \{\pm 1, \pm 3, \ldots, \pm m, 0\}$ such as to fully utilize the fast memory. (See [11, 23, 16] for details regarding left-to-right signed fractional window conversions.)

We have repeatedly simulated the behavior of our technique for uniformly random exponents in the interval $(0, \ldots, 2^{\ell} - 1)$, covering both the case of "new bases" to create cache entries (Section 3.1) and the case of "old bases" to observe the performance given such cache entries (Section 3.2). In these simulations, we have counted the following operations:

- Squarings (S) and other multiplications (M) used for precomputing powers of g_1 (including powers of cache entries derived from g_1);
- squarings (S) and multiplications (M) by precomputed powers of g_1 (or of cache entries) within the interleaved multi-exponentiation algorithm.

We have excluded from counting any of the multiplications by fixed precomputed elements (from ROM), since these are not a limiting factor given the assumption that plenty of space is available for these elements: low-weight exponent representations accordingly may be used for the corresponding exponents, so changes of the parameterization have less of an impact here. (Refer to Section 2.1 for applicable weight estimates.)

The simulation results can be found in Table 1. The values in the first row ($s = 1$) reflect the special situation when no splitting at all is done. This applies to the multi-exponentiation algorithm for a new base g_1 for which no cache entry is available (Section 3.1), where a signed-digit representation is used for the full-length exponent e_1. The remaining rows contain operation counts for cases where g_1 is an old base, i.e., an existing cache entry is used (Section 3.2). As we can see from the table, the number of squarings will be reduced to about ℓ/s as expected using the new modular exponent splitting technique. Moreover, the number of multiplications performed during the multi-exponentiation slightly increases from row to row: this due to the fact that smaller digit sets have to be

Table 1. Experimental performance figures (squarings and multiplications with powers of g_1) for s-fold exponent splitting with exponents up to ℓ-bits, with space for 4, 8, 16, or 32 elements for variable precomputation.

		$\#var = 4$	$\#var = 8$	$\#var = 16$	$\#var = 32$
$s = 1$	precomp.	$1S + 3M$	$1S + 7M$	$1S + 15M$	$1S + 31M$
	$\ell = 160$	$159.9S + 31.5M$	$156.0S + 26.1M$	$155.0S + 22.3M$	$154.0S + 19.5M$
	$\ell = 192$	$188.9S + 37.9M$	$187.9S + 31.4M$	$187.0S + 26.9M$	$186.0S + 23.5M$
	$\ell = 256$	$252.9S + 50.6M$	$251.9S + 42.1M$	$251.0S + 36.0M$	$250.0S + 31.5M$
$s = 2$	precomp.	$2S + 2M$	$2S + 6M$	$2S + 14M$	$2S + 30M$
	$\ell = 160$	$79.5S + 39.9M$	$79.1S + 32.0M$	$78.6S + 26.8M$	$78.6S + 23.2M$
	$\ell = 192$	$95.7S + 47.9M$	$94.7S + 38.5M$	$95.0S + 32.2M$	$93.7S + 27.8M$
	$\ell = 256$	$127.6S + 63.9M$	$126.9S + 51.4M$	$126.6S + 42.8M$	$126.8S + 36.8M$
$s = 3$	precomp.	$1S + 1M$	$3S + 5M$	$3S + 13M$	$3S + 29M$
	$\ell = 160$	$54.7S + 49.4M$	$53.4S + 37.4M$	$52.8S + 30.5M$	$52.4S + 25.9M$
	$\ell = 192$	$64.3S + 59.1M$	$63.3S + 44.7M$	$62.9S + 36.6M$	$62.3S + 31.1M$
	$\ell = 256$	$85.8S + 78.6M$	$85.0S + 59.6M$	$84.5S + 48.5M$	$84.5S + 41.1M$
$s = 4$	precomp.	$0S + 0M$	$4S + 4M$	$4S + 12M$	$4S + 28M$
	$\ell = 160$	$40.6S + 53.9M$	$40.7S + 40.7M$	$39.3S + 32.8M$	$38.8S + 27.8M$
	$\ell = 192$	$48.4S + 64.5M$	$47.6S + 48.6M$	$47.6S + 39.2M$	$46.9S + 33.1M$
	$\ell = 256$	$64.1S + 85.7M$	$64.1S + 64.7M$	$63.4S + 52.1M$	$62.9S + 43.8M$
$s = 5$	precomp.		$3S + 3M$	$5S + 11M$	$5S + 27M$
	$\ell = 160$		$33.0S + 46.4M$	$31.9S + 36.0M$	$31.1S + 30.0M$
	$\ell = 192$		$39.7S + 55.8M$	$39.4S + 43.1M$	$38.4S + 35.7M$
	$\ell = 256$		$51.8S + 73.7M$	$51.4S + 56.9M$	$50.5S + 47.1M$
$s = 6$	precomp.		$2S + 2M$	$6S + 10M$	$6S + 26M$
	$\ell = 160$		$29.4S + 50.4M$	$29.0S + 38.8M$	$27.8S + 31.9M$
	$\ell = 192$		$32.0S + 59.6M$	$32.2S + 45.9M$	$31.7S + 37.7M$
	$\ell = 256$		$45.0S + 79.7M$	$45.7S + 60.9M$	$44.3S + 49.8M$
$s = 7$	precomp.		$1S + 1M$	$7S + 9M$	$7S + 25M$
	$\ell = 160$		$27.8S + 53.8M$	$26.9S + 40.8M$	$26.0S + 33.5M$
	$\ell = 192$		$30.1S + 64.0M$	$28.9S + 48.6M$	$28.4S + 39.5M$
	$\ell = 256$		$40.5S + 84.7M$	$39.4S + 64.1M$	$38.2S + 52.1M$

used to obey the space limits while the splitting parameter is increased. (Note that $s \geq 5$ cannot be used with space for only 4 dynamically precomputed elements, so the corresponding parts of the table are left empty.)

Note that the size of cache entries does not affect the statistics as reflected in the table. With severe memory constraints for the cache, $s = 2$ might be the only option. Comparing the row $s = 1$ (which describes the case of a multi-exponentiation not using a cache entry) with the row $s = 2$ shows that our technique provides for a noticeable speed-up even with just $s = 2$.

It also should be noted that our multi-exponentiation technique for old bases g_1 (Section 3.2) involves $s - 1$ divisions with remainder to perform s-fold modular exponent splitting. This starts with an ℓ-bit denominator and a divisor around $\ell - \frac{\ell}{s}$ bits; both operands will decrease by around $\frac{\ell}{s}$ in each subsequent division. Thus, the total extra cost of these modular divisions should usually be

reasonably small. The typical size of results means that around ℓ bits will still suffice to store the resulting shorter exponents.

Please refer to Appendix B for certain implementation aspects. See Appendix C for a performance comparison of our technique with an immediate approach in a particular scencario.

References

1. American National Standards Institute (ANSI). Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA). ANSI X9.62, 1998.
2. Antipa, A., Brown, D., Gallant, R., Lambert, R., Struik, R., and Vanstone, S. Accelerated verification of ECDSA signatures. In *Selected Areas in Cryptography – SAC 2005* (2006), B. Preneel and S. Tavares, Eds., vol. 3897 of *Lecture Notes in Computer Science*, pp. 307–318.
3. Avanzi, R. M. A note on the sliding window integer recoding and its left-to-right analogue. In *Selected Areas in Cryptography – SAC 2004* (2005), H. Handschuh and M. A. Hasan, Eds., vol. 3357 of *Lecture Notes in Computer Science*, pp. 130–143.
4. Bernstein, D. J. Pippenger's exponentiation algorithm. Draft, 2002. Available from `http://cr.yp.to/papers.html#pippenger`.
5. Brickell, E. F., Gordon, D. M., McCurley, K. S., and Wilson, D. B. Fast exponentiation with precomputation. In *Advances in Cryptology – EUROCRYPT '92* (1993), R. A. Rueppel, Ed., vol. 658 of *Lecture Notes in Computer Science*, pp. 200–207.
6. Certicom Research. Standards for efficient cryptography – SEC 2: Recommended elliptic curve cryptography domain parameters. Version 1.0, 2000. Available from `http://www.secg.org/`.
7. Cohen, H., Ono, T., and Miyaji, A. Efficient elliptic curve exponentiation using mixed coordinates. In *Advances in Cryptology – ASIACRYPT '98* (1998), K. Ohta and D. Pei, Eds., vol. 1514 of *Lecture Notes in Computer Science*, pp. 51–65.
8. de Rooij, P. Efficient exponentiation using precomputation and vector addition chains. In *Advances in Cryptology – EUROCRYPT '94* (1995), T. Helleseth, Ed., vol. 950 of *Lecture Notes in Computer Science*, pp. 389–399.
9. Gordon, D. M. A survey of fast exponentiation methods. *Journal of Algorithms 27* (1998), 129–146.
10. Institute of Electrical and Electronics Engineers (IEEE). IEEE standard specifications for public-key cryptography. IEEE Std 1363-2000, 2000.
11. Khabbazian, M., and Gulliver, T. A. A new minimal average weight representation for left-to-right point multiplication methods. Cryptology ePrint Archive Report 2004/266, 2004. Available from `http://eprint.iacr.org/`.
12. Lim, C. H., and Lee, P. J. More flexible exponentiation with precomputation. In *Advances in Cryptology – CRYPTO '94* (1994), Y. G. Desmedt, Ed., vol. 839 of *Lecture Notes in Computer Science*, pp. 95–107.
13. Miyaji, A., Ono, T., and Cohen, H. Efficient elliptic curve exponentiation. In *International Conference on Information and Communications Security – ICICS '97* (1997), Y. Han, T. Okamoto, and S. Qing, Eds., vol. 1334 of *Lecture Notes in Computer Science*, pp. 282–290.
14. Möller, B. Algorithms for multi-exponentiation. In *Selected Areas in Cryptography – SAC 2001* (2001), S. Vaudenay and A. M. Youssef, Eds., vol. 2259 of *Lecture Notes in Computer Science*, pp. 165–180.

15. Möller, B. Improved techniques for fast exponentiation. In *Information Security and Cryptology – ICISC 2002* (2003), P. J. Lee and C. H. Lim, Eds., vol. 2587 of *Lecture Notes in Computer Science*, pp. 298–312.
16. Möller, B. Fractional windows revisited: Improved signed-digit representations for efficient exponentiation. In *Information Security and Cryptology – ICISC 2004* (2005), C. Park and S. Chee, Eds., vol. 3506 of *Lecture Notes in Computer Science*, pp. 137–153.
17. Muir, J. A., and Stinson, D. R. New minimal weight representations for left-to-right window methods. In *Topics in Cryptology – CT-RSA 2005* (2005), A. Menezes, Ed., vol. 3376 of *Lecture Notes in Computer Science*, pp. 366–383.
18. Muir, J. A., and Stinson, D. R. Minimality and other properties of the width-w nonadjacent form. *Mathematics of Computation 75* (2006), 369–384.
19. National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). FIPS PUB 186-2, 2000.
20. Okeya, K., Schmidt-Samoa, K., Spahn, C., and Takagi, T. Signed binary representations revisited. In *Advances in Cryptology – CRYPTO 2004* (2004), M. Franklin, Ed., vol. 3152 of *Lecture Notes in Computer Science*, pp. 123–139.
21. Pippenger, N. The minimum number of edges in graphs with prescribed paths. *Mathematical Systems Theory 12* (1979), 325–346.
22. Pippenger, N. On the evaluation of powers and monomials. *SIAM Journal on Computing 9* (1980), 230–250.
23. Schmidt-Samoa, K., Semay, O., and Takagi, T. Analysis of fractional window recoding methods and their application to elliptic curve cryptosystems. *IEEE Transactions on Computers 55* (2006), 48–57.
24. Struik, M., Brown, D. R., Vanstone, S. A., Gallant, R. P., Antipa, A., and Lambert, R. J. Accelerated verification of digital signatures and public keys. United States Patent Application Publication US 2007/0064932 A1, 2007.

A An Example

As an illustrative toy example, let us apply our new technique to multi-exponentiations $g_1^d \cdot g_2^e$ with exponents of size at most $\ell = 18$ bits. To keep the example simple, we use unsigned-digit (instead of signed-digit) exponent representations. Let the digit sets for the fixed base be $B_2 = \{1, 3, 5, 7, 0\}$. For radix-2 and modular exponent splitting, we use splitting parameter $s = 3$. Thus, g_2 is replaced a priori by three fixed bases g_2, g_3, g_4 where $g_3 = g_2^{2^6}, g_4 = g_2^{2^{12}}$. Accordingly, we precompute the powers

$$(g_2, g_2^3, g_2^5, g_2^7,\ g_3, g_3^3, g_3^5, g_3^7,\ g_4, g_4^3, g_4^5, g_4^7)$$

and save this data in ROM. We consider an environment with a limited amount of fast read/write memory and assume that we have only space to store 8 powers of the variable base g_1. Hence, we can choose digit set $B_1 = \{1, 3, \ldots, 15, 0\}$ for exponentiations with a new base (Section 3.1) and digit sets $B_1 = \{1, 3, 0\}, B_{G_1} = B_{G_2} = \{1, 3, 5, 0\}$ for exponentiations with an old base (Section 3.2).

Multi-Exponentiation for a New Base g_1. Let us now consider the computation of $g_1^d \cdot g_2^e$ for $d = 205802 = (110010001111101010)_2$ and $e = 245153 = (111011110110100001)_2$ where g_1 is a new base, i.e. no cached precomputed data based on g_1 is available. Before the actual multi-exponentiation, we compute the powers

$$(g_1, g_1^3, \ldots, g_1^{15})$$

and save these in fast read/write memory. Encoding $e_1 := d$ using B_1 yields

$$e_1 = (3, 0, 0, 1, 0, 0, 0, 0, 0, 0, 15, 0, 0, 5, 0, 1, 0)_2.$$

Encoding e using B_2 and then splitting into three parts e_2, e_3, e_4 yields

$$\begin{aligned} e_4 &= \quad (7, 0, 0, 0)_2, \\ e_3 &= (7, 0, 0, 5, 0, 0)_2, \\ e_2 &= (5, 0, 0, 0, 0, 1)_2. \end{aligned}$$

The following table shows what happens while performing the multi-exponentation $\prod_{i=1}^{4} g_i^{e_i}$ as described in Section 3.1, based on interleaved multi-exponentiation as explained in Section 2.1:

j	A	Cache entry (so far)
17	1	(g_1)
16	g_1^3	
13	$(g_1^3)^{2^3} g_1 = g_1^{25}$	
12	$(g_1^{25})^2 = g_1^{50}$	$\big(g_1, (50, g_1^{50})\big)$
6	$(g_1^{50})^{2^6} g_1^{15} = g_1^{3215}$	$\big(g_1, (3215, g_1^{3215}), (50, g_1^{50})\big)$
5	$(g_1^{3215})^2 g_2^5 g_3^7 = g_1^{6430} g_2^5 g_3^7$	
3	$(g_1^{6430} g_2^5 g_3^7)^{2^2} g_1^5 g_4^7 = g_1^{25725} g_2^{20} g_3^{28} g_4^7$	
2	$(g_1^{25725} g_2^{20} g_3^{28} g_4^7)^2 g_3^5 = g_1^{51450} g_2^{40} g_3^{61} g_4^{14}$	
1	$(g_1^{51450} g_2^{40} g_3^{61} g_4^{14})^2 g_1 = g_1^{102901} g_2^{80} g_3^{122} g_4^{28}$	
0	$(g_1^{102901} g_2^{80} g_3^{122} g_4^{28})^2 g_2 = g_1^{205802} g_2^{161} g_3^{244} g_4^{56}$	

As we can see here, until and including round $j = 6$, the variable A contains no powers of bases other than g_1. Intermediate powers of g_1 for caching are available at the points $j = 12$ and $j = 6$ of the computation.

Multi-Exponentiation for an Old Base g_1. Let us compute $g_1^d \cdot g_2^e$ for $d = 73660 = (10001111110111100)_2$, $e = 236424 = (111001101110001000)_2$ where g_1 is an old base for which the cache entry

$$(g_1, (\lambda_1 = 3215, G_1 = g_1^{3215}), (\lambda_2 = 50, G_2 = g_1^{50}))$$

as created above is available. First, the powers

$$(g_1, g_1^3,\ G_1, G_1^3, G_1^5,\ G_2, G_2^3, G_2^5)$$

are precomputed and stored in fast read/write memory. Next, we perform modular exponent splitting as described in Section 3.2:

$$\begin{aligned}
&d_0 = d = 73660,\\
&E_1 = \left\lfloor \tfrac{d_0}{\lambda_1} \right\rfloor = 22 \text{ and } d_1 = d_0 - E_1\lambda_1 = 2930,\\
&E_2 = \left\lfloor \tfrac{d_1}{\lambda_2} \right\rfloor = 58 \text{ and } d_2 = d_1 - E_2\lambda_2 = 30,\\
&E_3 = d_2 = 30
\end{aligned}$$

Encoding E_1, E_2 and E_3 using B_{G_1}, B_{G_2} and B_1 yields

$$\begin{aligned}
E_1 &= (10110)_2 = (5,1,0)_2,\\
E_2 &= (111010)_2 = (3,0,0,5,0)_2,\\
E_3 &= (11110)_2 = (3,0,3,0)_2.
\end{aligned}$$

By encoding e using B_2 and then splitting into 3 parts e_2, e_3, e_4 (using radix-2 exponent splitting), we obtain

$$\begin{aligned}
e_4 &= (7,0,0,0)_2,\\
e_3 &= (3,0,0,0,7,0)_2,\\
e_2 &= (1,0,0,0)_2.
\end{aligned}$$

The table below shows what happens in the interleaved multi-exponentiation to compute $G_1^{E_1}G_2^{E_2}g_1^{E_3}g_2^{e_2}g_3^{e_3}g_4^{e_4}$:

j	A
5	g_3^3
4	$(g_3^3)^2G_2^3 = G_2^3g_3^6$
3	$(G_2^3g_3^6)^2g_1^3g_2g_4^7 = G_2^6g_1^3g_2g_3^{12}g_4^7$
2	$(G_2^6g_1^3g_2g_3^{12}g_4^7)^2G_1^5 = G_1^5G_2^{12}g_1^6g_2^2g_3^{24}g_4^{14}$
1	$(G_1^5G_2^{12}g_1^6g_2^2g_3^{24}g_4^{14})^2G_1G_2^5g_1^3g_3^7 = G_1^{11}G_2^{29}g_1^{15}g_2^4g_3^{55}g_4^{28}$
0	$(G_1^{11}G_2^{29}g_1^{15}g_2^4g_3^{55}g_4^{28})^2 = G_1^{22}G_2^{58}g_1^{30}g_2^8g_3^{110}g_4^{56}$

B Implementation Aspects

On-The-Fly Signed-Digit Conversions. In our descriptions of multi-exponentiation algorithms, we have employed radix-2 representations of exponents by referring to their individual digits. However, this by no means is meant to imply that these digits need to be explicitly obtained and stored in advance, which would be quite inconvenient if memory is scarce. Left-to-right signed fractional window representations [11, 23, 16] are very convenient for our purposes since (for any given maximum digit value m) there is a finite-state machine that transforms the binary representation into the corresponding signed-digit representation. As the name suggests, this conversion machine starts at the most significant digit ("left") and continues towards the least significant digit ("right"). Since interleaved multi-exponentiation is a left-to-right technique as well, this often means that the signed digits can be obtained on the fly.

To make this work with radix-2 exponent splitting, we need to add an additional first left-to-right pass through the binary representation. This is essentially

a dry run of the signed fractional window conversion, used to determine the first binary digits that will affect each of the segments of the signed-digit representation. For s-fold radix-2 exponent splitting, such a dry run can be used to initialize each of s finite-state machines, which afterwards can be used to obtain the digits of the individual segments (exactly as in the case of the on-the-fly conversion using just a single such machine that we would use in the case without splitting).

A simpler alternative would be to first split the binary representation, and then generate the signed-digit representations individually. This could be done truly on the fly, i.e., without the additional left-to-right pass. However, this strategy often will increase the total weight of the resulting representations [15], so the two-pass technique usually should lead to better performance.

Variants of the Signed Fractional Window Representation. In our performance estimates in Section 4, we optimistically assumed that besides ROM and fast read/write memory, there is another kind of memory that we can use for the cache. This is an assumption that we made for simplicity, but which is not necessary. In fact we may use some of the fast read/write memory for a small cache without completely losing this memory for precomputed powers of g_1.

This can be achieved by observing that we may modify the parameter m for the left-to-right signed fractional window representation *while performing the conversion.* Thus, in the algorithm from Section 3.1, provided that $m \geq 2s - 1$, we may initially use some maximum-size digit set $B^{\pm}(m) = \{\pm 1, \pm 3, \ldots, \pm m, 0\}$ for signed digits $b_{1,\ell}$ down to $b_{1,L_1+\cdots+L_{s-1}}$, then cache the current group element $g_1^{(b_{1,\ell},\ldots,b_{1,L_1+\cdots+L_{s-1}})_2}$ in the memory space that so far held g_1^m, and then use the smaller digit set $B^{\pm}(m-2)$ for subsequent digits $b_{1,L_1+\cdots+L_{s-1}-1}$ down to $b_{1,L_1+\cdots+L_{s-2}}$. Continuing in this fashion, we eventually give up digits $\pm m$, $\pm(m-2)$, $\ldots$, $\pm\big(m-2(s-1)\big)$.

C Performance Comparison

This appendix demonstrates the merits of our new technique for multi-exponentiation with caching in one particular situation where very little memory is available for use as a cache. We show that our method is of advantage even under this severe restriction. We make the following assumptions:

- We look at two-fold multi-exponentiation, $g_1^{e_1} g_2^{e_2}$. Base element g_2 is fixed; base element g_1 is variable such that the current value will repeat in the directly following multi-exponentiation with probability $P_{\text{old}} = \frac{1}{2}$.
- The exponents e_1 and e_2 are uniformly random integers up to $\ell = 256$ bits.
- Storage is available for 128 fixed precomputed elements in read-only memory (derived from the fixed base g_2), and for only 2 precomputed elements in read/write memory. The latter includes input value g_1. In addition to this, we have space for variable A in the algorithm from Section 2.1, and the memory holding the exponents. (Note that typically the memory needed for the exponents is less than the memory needed for a single group element: for

elliptic curve cryptography using projective coordinates over a 256-bit field, one group element takes 768 bits.)
- Different from the assumptions as used in Section 4, we have *no additional cache memory.* That is, a group element to be cached has to be kept in one of the two read/write storage units for precomputed elements.
- We use rough estimates $S = 0.7$ and $M = 1$ for the amount of time spent on each group squaring (e.g., elliptic curve point doubling) and on each group multiplication (e.g., elliptic curve point addition). (For example, when using Jacobian projective coordinates for elliptic curves over prime fields, a point doubling takes 10 or 8 field multiplications depending on the curve, and a general point addition requires 16 field multiplications [10], or 11 field multiplications in the case of "mixed coordinates" [7]. Mixed coordinates require a one-time conversion step to one of the inputs to convert it into affine coordinates, which is reasonable for precomputed values. Accordingly, $\frac{8}{11} \approx 0.73$ is one way to justify our estimate $\frac{S}{M} \approx 0.7$, although in the following we neglect the cost of the conversion.)

If (instead of applying our new caching strategy) we directly use interleaved multi-exponentiation in this situation, employing signed-digit representation as explained in Section 2.1, we can keep precomputed values $g_2, g_2^3, g_2^5, \ldots, g^{255}$ in read-only memory, and use read/write memory for g_1 and g_1^3, thus achieving an exponentiation cost of approximately

$$\Big(\frac{256}{4} + \frac{256}{10}\Big)M + 255S \approx 268.1$$

(or $89.5M + 254S \approx 267.3$ according to experimental simulation results) plus $1M+1S = 1.7$ to precompute g_1^3 from g_1 when g_1 has changed from the previous computation. By assumption, this happens with probability $\frac{1}{2}$, resulting in a total estimate of $268.1 + \frac{1.7}{2} \approx 269.0$ (for the simulation: 268.2).

Our method initially performs worse than this, namely, in the case with a new base (Section 3.1). Here, the read-only memory will contain $g_2, g_2^3, g_2^5, \ldots, g^{127}$, plus similar powers of $g_2^{2^{128}}$. The read/write memory initially is filled with precomputed elements g_1 and g_1^3. To perform the multi-exponentiation as described in Section 3.1, we use radix-2 exponent splitting for exponent e_2 to obtain partial exponent representations no longer than 129 digits. For exponent e_1, we use a signed fractional window representation variant as sketched in Appendix B, i.e., where digit set parameter m is modified within the conversion: the more significant digits can use digits set $\{\pm 1, \pm 3, 0\}$, whereas the less significant digits (digits $b_{1,127}, \ldots b_{1,0}$) are restricted to digit set $\{\pm 1, 0\}$. This is because we no longer keep g_1^3 in memory when the method from Section 3.1 has determined a group element to be cached, thus freeing a memory location for use as cache space. The performance estimate for this multi-exponentiation is

$$\Big(\frac{128}{4} + \frac{128}{3} + \frac{256}{9}\Big)M + 255S \approx 281.6$$

(simulation: $102.7M + 253.8S \approx 280.4$) plus $1M + 1S \approx 1.7$ to precompute g_1^3 from g_1. We benefit from the extra effort put into this computation whenever

the same g_1 reappears in the following multi-exponentiation. In this case, the multi-exponentiation will only take approximate effort

$$\Big(\frac{128}{3} + \frac{128}{3} + \frac{256}{9}\Big)M + 127S \approx 202.7$$

(simulation: $113.9M + 128.2S \approx 203.6$). The average cost given $P_{\text{old}} = \frac{1}{2}$ comes to 242.1 (simulation: 242.0). Thus, our method provides an average 10 percent performance improvement in this scenario.