Efficient Java Implementation of Elliptic Curve Cryptography for J2ME-Enabled Mobile Devices

Johann Großschädl[1], Dan Page[2], and Stefan Tillich[2]

[1] University of Luxembourg, CSC Research Unit, LACS,
6, rue Richard Coudenhove-Kalergi, L–1359 Luxembourg, Luxembourg
johann.groszschaedl@uni.lu
[2] University of Bristol, Department of Computer Science,
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, U.K.
{page,tillich}@cs.bris.ac.uk

Abstract. The Micro Edition of the Java 2 platform (J2ME) provides an application environment specifically designed to address the demands of embedded devices like cell phones, PDAs or set-top boxes. Since the J2ME platform does not include a crypto package, developers are forced to use third-party classes or to implement all cryptographic primitives from scratch. However, most existing implementations of elliptic curve (EC) cryptography for J2ME do not perform well on resource-restricted devices, in most cases due to poor efficiency of the underlying arithmetic operations. In this paper we present an optimized Java implementation of EC scalar multiplication that combines efficient finite-field arithmetic with efficient group arithmetic. More precisely, our implementation uses a pseudo-Mersenne (PM) prime field for fast modular reduction and a Gallant-Lambert-Vanstone (GLV) curve with an efficiently computable endomorphism to speed up the scalar multiplication with random base points. Our experimental results show that a conventional mobile phone without Java acceleration, such as the Nokia 6610, is capable to execute a 174-bit scalar multiplication in roughly 400 msec, which is more than 45 times faster than the widely-used Bouncy Castle Lightweight Crypto API for J2ME.

1 Introduction

The Java programming language, introduced by Sun Microsystems in 1995, was originally designed to simplify the software engineering for consumer electronics [13, 2]. Many characteristics of the Java language stem from the focus towards the consumer marketplace with its vast number of different hardware platforms and (largely incompatible) operating systems. Unlike C or C++, Java is generic and *platform-independent* because it is an interpreted rather than a compiled language. That is, when Java source code is compiled, it is not compiled into architecture-dependent "machine" instructions, but into an architecture-neutral intermediate representation consisting of generic instructions (called *bytecode*) to be executed by a *Java Virtual Machine (JVM)* [7]. The JVM can be seen as a program running on the host processor that interprets generic Java bytecodes

and translates them to the processor's native machine instructions as they are executed. A program written in Java and compiled to generic bytecode will, in principle, run unchanged on any platform for which a JVM exists, regardless of what operating system or processor lies underneath. This "Write Once, Run Everywhere" capability has made Java the de-facto standard language for the development of cross-platform applications in the embedded domain.

The Java 2 platform is available in three different editions, each aimed at a specific area of application: J2EE (Java 2 Enterprise Edition) for developing and deploying large-scale server applications, J2SE (Java 2 Standard Edition) for the implementation of Java programs that can be executed on commodity computers, as well as *J2ME (Java 2 Micro Edition)* for creating Java applications to be run on various kinds of mobile and embedded devices[3]. These platform editions differ in terms of size (and complexity) of the class library and the capabilities of the corresponding JVM. J2ME is, roughly speaking, a stripped-down version of J2SE that contains only a small subset of the standard Java class library, in addition to J2ME-specific classes [28]. Java applications for mobile phones are called *midlets* and can be run on devices on which at least 192 kB of memory is available to the J2ME platform. Since the "conventional" interpretation of Java bytecode on such resource-restricted devices is rather slow, different techniques for speeding up the execution of midlets have emerged. For example, some JVMs feature a *Just-In-Time (JIT)* compiler that translates frequently-executed code segments ("hot spots") into native machine code at run-time (i.e. during execution of the midlet) [8]. Another approach is the provision of dedicated hardware support for the JVM through architectural extensions that allow the underlying processor to directly execute Java bytecode (e.g. ARM's Jazelle [1]).

Since J2ME is, roughly speaking, a stripped-down version of J2SE for mobile devices with restricted resources, it lacks many of the classes found in the class library of the larger editions. While the Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE) are included in J2SE/J2EE, they are absent in J2ME for both technical and legal reasons [28]. Therefore, developers of J2ME applications are forced to use third-party classes or to implement the required cryptographic operations from scratch. The latter is a tedious task, especially for Elliptic Curve (EC) cryptography, since the J2ME class library does not even contain the J2SE `BigInteger` class for multi-precision arithmetic. On the other hand, most existing EC implementations for J2ME (e.g. the Bouncy Castle Lightweight API [26] or the SIC JCE-ME [25]) do not perform very well since they aim to provide high flexibility (i.e. support of many implementation options) rather than high speed. There exist only a few performance-optimized Java implementations of EC cryptography, which is surprising given that more than three billion J2ME-enabled mobile phones have shipped to date[4]. Also the scientific literature on optimizing EC cryptosystems for the J2ME platform is

[3] In June 2005, Sun Microsystems renamed J2EE to Java EE, J2SE to Java SE, and J2ME to Java ME. However, since the old names are still very common and widely used today, we decided to stick with the term "J2ME" in this paper.

[4] `http://www.java.com/en/about`, `http://java.sun.com/products/javadevice`

sparse compared to the vast number of papers presenting EC implementations written in C and/or Assembly language.

In the following sections, we describe our efforts to develop a performance-optimized Java implementation of EC scalar multiplication for J2ME-enabled devices. The focus of our work was to achieve fast execution time, low memory (i.e. RAM) requirements, and small bytecode size. After a thorough evaluation of different implementation options, we decided to use a pseudo-Mersenne (PM) prime field [9] over which a Gallant-Lambert-Vanstone (GLV) curve with good cryptographic properties can be defined [12, 15]. This particular choice of type of field and type of curve allows for combining fast modular arithmetic (due to the special form of the prime) with fast EC point arithmetic (thanks to an efficiently computable endomorphism). To exemplify our approach, we present an optimized Java implementation of EC scalar multiplication on the GLV curve $y^2 = x^3 - 7$ over the PM prime field $\mathbb{F}_p$ with $p = 2^{174} - 3$. The group of points on this curve has prime cardinality and offers a security level of 87 bits[5]. When using mixed Jacobian-affine coordinates, a point addition on this curve requires eight multiplications (8M) and three squarings (3S) in the underlying field (see [15] for the exact formula). The double of a point given in Jacobian coordinates can be computed using only 3M and 4S since the curve parameter a is 0. A full scalar multiplication of an arbitrary point P by an n-bit integer k represented in Non-Adjacent Form (NAF) [15] takes $5.\dot{6}n$ multiplications and $5n$ squarings on average, i.e. $5.\dot{6}\mathrm{M} + 5\mathrm{S}$ per bit. However, the cost of a scalar multiplication on our GLV curve can be significantly reduced by exploiting the efficiently-computable endomorphism described in [12, Ex. 4]. This endomorphism allows one to accomplish an n-bit scalar multiplication $k \cdot P$ through a computation of the form $k_1 \cdot P + k_2 \cdot Q$, whereby k_1, k_2 have only half the bit-length of k. The two half-length scalar multiplications can be performed simultaneously and require $n/2$ point doublings and roughly $n/4$ additions when k_1, k_2 are represented in Joint Sparse Form (JSF) [23]. Thus, the total cost of computing $k \cdot P$ amounts to $3.5n$ multiplications and $2.75n$ squarings in $\mathbb{F}_p$, i.e. $3.5\mathrm{M} + 2.75\mathrm{S}$ per bit.

Since our implementation aims at high performance, we did not attempt to make the scalar multiplication on our GLV curve resistant against side-channel attacks such as Simple Power Analysis (SPA) [20]. However, all arithmetic operations (except inversion) in the underlying 174-bit prime field are written in a highly regular fashion without any conditional statements, i.e. branches. Such a branch-less implementation of the field arithmetic helps the VM to identify the performance-critical code sections for JiT-compilation and also eliminates the performance penalty due to branch mis-predictions. Note that, even though the field-arithmetic is SPA-resistant, our implementation of scalar multiplication is vulnerable to SPA attacks. We will address this issue in future work dedicated to SPA countermeasures for scalar multiplication on GLV curves.

[5] More precisely, the cardinality of the group of $\mathbb{F}_p$-rational points on said curve is a 174-bit prime. A properly implemented EC cryptosystem using this group provides a security level similar to that of a 87-bit secret-key cryptosystem, which is well above the "smallest general-purpose level" of 80 bits recommended by ECRYPT [22].

2 Prime-Field Arithmetic in Java

Our optimized Java software for EC scalar multiplication uses the prime field $\mathbb{F}_p$ of order $p = 2^{174} - 3$ as underlying algebraic structure. In the following, we explain the rationale behind choosing this particular field and elaborate on the efficient implementation of multiple-precision arithmetic in Java. Most previous software implementations of EC cryptography were written in C (or C++) and contain hand-optimized Assembly code for the performance-critical arithmetic operations. However, even though C/C++ and Java have a similar syntax, there exist also some differences which impact the implementation and optimization of multiple-precision arithmetic.

2.1 Selection of Prime Field

A major aspect when implementing EC cryptography is to find an appropriate trade-off between performance and security; this is particularly important for a Java implementation to be run on J2ME-enabled mobile phones or PDAs since (1) such devices are constrained in processing power and (2) the interpretation of platform-independent byte code is much slower than the execution of native machine code. The ECRYPT II report on algorithms and keysizes considers 80 bits as the smallest level of security that "protects against the most reasonable and threatening attack (key search) scenarios" [22, p. 32]. An 80-bit symmetric key can, under certain assumptions, be seen "equivalent" to an EC key of size 160 bits [22, Table 7.2]. In order to support 160-bit keys, an EC cryptosystem must be designed on basis of an EC group of order roughly 2^{160}, which, due to the Hasse-Weil theorem [5, page 278], requires an underlying field of (at least) 160 bits. However, GLV curves have a special structure that could reduce the time required to compute EC discrete logarithms by "a small factor" [12, Section 5], similar to Koblitz curves [29]. To account for this structure, we have to slightly increase the order of the EC group and, consequently, the order of the underlying prime field, e.g. to around 170 bits.

A second criterion to consider when choosing a prime field $\mathbb{F}_p$ for EC cryptography is the efficiency of arithmetic operations in this field, in particular the efficiency of the reduction modulo p. Mersenne primes are special primes of the form $p = 2^k - 1$ and allow for particularly fast implementation of the reduction operation. A $2k$-bit number x can be reduced modulo $p = 2^k - 1$ by adding the higher half of x (i.e. the k most significant bits of x) to the lower half, followed by conditional subtraction(s) of p to obtain the least non-negative residue. Consequently, the reduction costs merely an addition of two k-bit numbers modulo p. Unfortunately, primes of the form $2^k - 1$ are very rare; none of the Mersenne numbers between $2^{127} - 1$ and $2^{521} - 1$ is prime [5]. The second-best option in terms of efficiency of the modular reduction are *pseudo-Mersenne (PM)* primes [9], i.e. primes that can be written as $p = 2^k - c$ where c is "small" compared to 2^k. In the ideal case $c = 3$, which allows a reduction operation to be carried out through three k-bit additions modulo p. A prime of the form $2^k - 3$ with a length of roughly 170 bits exists, namely $p = 2^{174} - 3$.

A third requirement on the prime p is the suitability to define a GLV curve with "good" cryptographic properties over the field $\mathbb{F}_p$. GLV curves of the form $y^2 = x^3 + ax$ require the underlying prime field $\mathbb{F}_p$ to contain an element of order 4 (which is the case if $p \equiv 1 \bmod 4$), whereas GLV curves defined via the equation $y^2 = x^3 + b$ need a field $\mathbb{F}_p$ with $p \equiv 1 \bmod 3$ [12]. Furthermore, the curve itself (or, more precisely, the group of points on the curve) has to satisfy certain properties, e.g. it has to contain a large subgroup of prime order. The GLV curve $y^2 = x^3 - 7$ over the 174-bit prime field specified above fulfills this property and also meets a number of other security criteria as we will show in more detail in Subsection 3.1. Taking all these considerations into account, we decided to use the prime field $\mathbb{F}_p$ with $p = 2^{174} - 3$ for our Java implementation of EC scalar multiplication.

Representation of Field Elements. State-of-the-art cryptographic libraries represent the elements of a large prime field (i.e. long integers) as arrays of single-precision words, e.g. arrays of type `unsigned int`. Most high-performance implementations written in C or Assembly language match the number of bits per single-precision word to the word-size of the target processor so as to take advantage of the full length of the datapath. Also the J2SE `BigInteger` class follows a similar idea; it uses an `int`-array as internal data structure and stores 32 bits in each element of the array. However, this approach yields sub-optimal results when the size (i.e. bitlength) of the prime field is constant and known in advance. These inefficiencies, which will be discussed in more detail below, are due to the fact that Java is a platform-independent programming language and that it does not support unsigned data types.

We conducted experiments with different representations of the $\mathbb{F}_p$-elements and found that splitting the 174-bit integers into six 29-bit words allows one to achieve the best performance. Our implementation does not store these words in an array but in six independent variables of type `int`, which are declared as `private` within the Java class that performs the $\mathbb{F}_p$-arithmetic so that they are not visible (and accessible) from "outside" [2]. Avoiding the use of an array is possible and viable in our case since the length of the operands is fixed and we implemented all arithmetic operations (except inversion) in an unrolled fashion without executing any conditional statements to achieve a maximum of performance and resistance against side-channel attacks. Formally, our implementation of $\mathbb{F}_p$-arithmetic uses a number representation radix of 2^{29}, which means that any integer $a \in \mathbb{F}_p$ can be written as

$$a = \sum_{i=0}^{5} a_i \cdot 2^{29i} \quad \text{with} \quad 0 \leq a_i \leq 2^{29} - 1. \tag{1}$$

However, similar to the work of Bernstein [4], we allow (i.e. tolerate) individual words a_i slightly larger than the radix of 2^{29} for reasons of both efficiency and security. Furthermore, our implementation allows incompletely reduced results in the sense that an arithmetic operation does not necessarily return the least non-negative residue modulo p, but the result is always in the range $[0, 2p]$.

2.2 Efficient Arithmetic Modulo $p = 2^{174} - 3$

In the following, we describe our Java implementation of fast arithmetic in the prime field $\mathbb{F}_p$, i.e. addition and multiplication modulo $p = 2^{174} - 3$. We will in particular explain the efficiency of our 29-bit-per-word representation of field elements versus a straightforward representation using 32-bit words.

Addition and Subtraction. Most high-speed cryptographic libraries written in C (or C++) use small Assembly-code fragments to implement performance-critical operations such as multiple-precision addition and multiplication. The former operation can be realized in a (relatively) simple yet efficient way if the underlying processor provides an add-with-carry instruction. Even though also Java features a mechanism to include Assembly code, namely the Java Native Interface (JNI), we did not follow this approach because it contradicts the idea of platform-independence. Furthermore, one has to take into account that some processors (e.g. MIPS, Alpha) do not possess an add-with-carry instruction. To perform multi-precision addition on these processors, the add-with-carry has to be emulated, e.g. by first executing the ordinary `add` instruction, followed by a comparison of the result with one of the two operands. If the obtained sum is smaller than either of the operands, then an overflow occurred, i.e. the addition produced a carry, which must be processed properly when adding the two next-higher words. Consequently, from the second word-pair onwards, the addition becomes more complex and time-consuming since an "incoming" carry must be considered.

High-level language implementations of multi-precision addition can follow a very similar approach as described above if the long integers are represented in such a way that the number of bits per single-precision word corresponds to the word-size of the underlying processor (e.g. 32 bits per word on a 32-bit processor). However, the performance of multi-precision addition in C/C++ or Java (or in Assembly language on processors without add-with-carry instruction) can be greatly improved by reducing the number of bits per word, e.g. to 31 when working on a 32-bit machine. In this case, the sum of two single-precision words is at most 32 bits long and, hence, fits into a 32-bit word, i.e. an overflow can not occur. The carry bit is stored in the MSB of the result-word and accessible through a right-shift by 31 bit positions. This approach was originally proposed roughly 30 years ago by Hennessy et al [16, Section 2.3.3] to demonstrate the ability of the MIPS architecture (and other architectures lacking a carry flag) to efficiently perform multi-precision addition.

Algorithm 1 shows our Java implementation of addition in $\mathbb{F}_p$; we follow, in principle, the approach of Hennessy et al [16] described above, except that we represent the long integers using 29-bit words. The variable s, which is of type `int` in our implementation, holds the sum of two 29-bit operand-words plus the carry bit. Even though the sum can have a length of up to 30 bits, only the 29 LSBs of s are actually assigned to the result-word z_i. The three MSBs of s are masked out via a logical AND operation using a mask value of 0x1fffffff (see line 4 of Algorithm 1). Before adding the next pair of operand-words, the sum

Algorithm 1. Addition in $\mathbb{F}_p$

Input: Two 174-bit integers, $A = (a_5, a_4, a_3, a_2, a_1, a_0)$ and $B = (b_5, b_4, b_3, b_2, b_1, b_0)$, represented by six 29-bit words.

Output: Modular sum $Z = A + B \bmod (2^{174} - 3) = (z_5, z_4, z_3, z_2, z_1, z_0)$.

1: $s \leftarrow 0$
2: **for** i from 0 by 1 to 5 **do**
3: $\quad s \leftarrow a_i + b_i + (s \gg 29)$
4: $\quad z_i \leftarrow s$ & 0x1fffffff
5: **end for**
6: $z_0 \leftarrow z_0 + 3 \cdot (s \gg 29)$
7: **return** $Z = (z_5, z_4, z_3, z_2, z_1, z_0)$

s is shifted 29 bit positions to the right to ensure the correct alignment between a_i, b_i, and the carry bit. In fact, the shift operation yields the carry bit from the previous addition of operand words, i.e. the value of the expression $(s \gg 29)$ in line 3 of Algorithm 1 is either 0 or 1. Note, however, that our implementation does not strictly adhere to the pseudo-code shown in Algorithm 1; we unrolled the loop to maximize performance.

The sum of two 174-bit integers can be up to 175 bits long, which means a reduction modulo $p = 2^{174} - 3$ may be necessary to obtain a final result within the range of $[0, p-1]$. Reducing the sum with respect to the prime p calls for a comparison between the sum and p, followed by a subtraction of p if the sum is greater than or equal to p. The former operation, when realized in a straightforward way, requires comparing up to six single-precision words. However, an exact comparison of the sum with the prime p can, in general, be avoided since modular arithmetic also works with incompletely reduced operands and, consequently, there is no need to fully reduce a result. Our implementation compares the sum with 2^{174} instead of the "exact" prime $p = 2^{174} - 3$, i.e. the result of a modular addition is not necessarily the least non-negative residue. The obvious advantage of performing the comparison in this way is efficiency; we just have to check whether the 174-bit addition produced a "carry-out," as described in [30, Section 3]. More precisely, after the final iteration of the loop in Algorithm 1, we simply right-shift the variable s by 29 bits to obtain the "carry out." The subtraction of the prime p is realized in our implementation by addition of the 174-bit two's complement of p to the sum, i.e. by addition of $2^{174} - p = 3$ to the least significant word z_0. Line 6 of Algorithm 1 performs the modular reduction operation, consisting of the extraction of the "carry out", its multiplication by 3, and the addition of the product to z_0. Note that, due to this addition, the least significant word z_0 can be up to 30 bits long. However, the extended size of z_0 does not require special consideration since all our arithmetic operations also work correctly with 30-bit words.

We implemented the subtraction of two elements $a, b \in \mathbb{F}_p$ through the operation $2p + a - b$, which, in turn, is carried out via word-level operations of the form $2p_i + a_i - b_i$, followed by a reduction of the result modulo p as described before. Performing the subtraction in this way (instead of a direct computation

Listing 1. Nested loop of multi-precision multiplication as implemented in the function `multiplyToLen` of the `BigInteger` class (version 1.76)

```
1  for (int i = xstart-1; i >= 0; i--) {
2      carry = 0;
3      for (int j=ystart, k=ystart+1+i; j>=0; j--, k--) {
4          long product = (y[j] & LONG_MASK) *
5                         (x[i] & LONG_MASK) +
6                         (z[k] & LONG_MASK) + carry;
7          z[k] = (int)product;
8          carry = product >>> 32;
9      }
10     z[i] = (int)carry;
11 }
```

of the difference $a - b$) ensures that the results of all word-level operations, as well as the final 174-bit result, are positive and can be calculated in a straightforward way without the need of any conditional statements. Such branch-less execution of arithmetic operations can help to prevent implementation attacks since always the same sequence of instructions is executed, independent of the actual value of the operands. In fact, due to loop unrolling, our implementation of subtraction in $\mathbb{F}_p$ does not execute any control-flow statements at all.

Multiplication and Squaring. A multiplication of two elements $a, b \in \mathbb{F}_p$ can be accomplished by conventional integer multiplication of a and b, along with a reduction of the obtained 384-bit product modulo the prime p. There exist two principal techniques for implementing multiple-precision multiplication in software, namely *operand scanning* (also called pencil-and-paper method [19]) and *product scanning* (also known as Comba's method [6]). Both methods execute the same number of single-precision multiplications, but differ in loop structure and inner-loop operation. The operand scanning technique uses a nested loop to calculate the double-precision partial products in a row-by-row fashion [14]. In each iteration of the inner loop, an operation of the form $a \cdot b + c + d$ is carried out, i.e. two single-precision words are multiplied together and two other words are added to the product. On the other hand, the product scanning algorithm is characterized by a nested-loop structure consisting of two outer loops and two inner loops; the first outer loop calculates the lower half of the product and the second outer loop the upper half [14]. The partial products are added up in a column-wise fashion using an inner-loop operation of the form $s + a \cdot b$, i.e. two words are multiplied and the product is added to a cumulative sum.

The relative performance of operand scanning vs. product scanning depends on a range of factors such as the programming language and the representation of the long integers. Most implementations of multiple-precision multiplication written in C/C++ or Java use the operand-scanning method and represent the

operands as arrays of single-precision words whose bitlength matches the wordsize of the underlying processor (e.g. 32-bit words on a 32-bit processor). The inner loop of the operand-scanning method is straightforward to implement in a high-level programming language since the result of an operation of the form $a \cdot b + c + d$ is at most 64 bits long (and, therefore, fits into a C/C++ variable of type `unsigned long long` or a Java variable of type `long`) when a, b, c, and d are 32-bit words. However, a peculiarity of Java, compared to C/C++, is the lack of unsigned data types, which calls for an implementation of the inner loop as shown in Listing 1. The variables `carry` and `product` are of type `long` and can hold 64-bit signed integers, whereas `x`, `y`, and `z` are 32-bit `int` arrays. Java requires a type conversion (or "widening") of the operands from `int` to `long` in order to get a 64-bit result when multiplying two 32-bit integers. An integral part of this conversion process is sign extension, which means that the upper 32 bits of the 64-bit `long` representation are filled with the sign bit of the original 32-bit `int` value. These upper 32 bits have to be masked out to obtain the corresponding unsigned value; the implementation shown in Listing 1 achieves this via a logical AND operation using `LONG_MASK`, a `final static` variable of type `long`, which is defined in the `BigInteger` class and initialized with the literal 0xffffffffL. Of course, performing three such maskings in the inner loop incurs a significant performance degradation.

The product scanning method, on the other hand, performs a multiply-accumulate operation of the form $s + a \cdot b$ in its inner loop, i.e. two single-precision words are multiplied and the double-precision product is added to a cumulative sum [6]. However, the bitlength of the cumulative sum can grow beyond double precision when several double-precision products are summed up, which makes the product scanning method quite hard to implement in C/C++ or Java since high-level languages do not provide a primitive integer type whose bitlength is more than twice the length of a single-precision word. A straightforward way to circumvent this problem is to reduce the number of bits per word to less than the wordsize of the underlying processor, following the rationale discussed earlier in this section for the addition of field elements. Since most J2ME-enabled mobile devices are equipped with 32-bit processors, we can, for example, use a 29-bit-per-word representation and implement the product scanning method as shown in Algorithm 2. Having single-precision words of 29 bits means that all double-precision partial products $a_j \cdot b_{i-j}$ in Algorithm 2 consist of at most 58 bits. Consequently, we can use a 64-bit variable of type `long` to hold the cumulative sum s; in this case, up to $2^{64-58} = 2^6 = 64$ partial products can be added up without overflowing s. Such a reduction of the bitlength of operand words (which was first suggested by Barrett [3, p. 317] roughly 25 years ago[6]) allows for a very efficient implementation of the inner loop since masking operations as in Listing 1 can be avoided.

The result of a multiplication performed according to the first part (i.e. line 1 – 16) of Algorithm 2 is a 348-bit product Z represented by an array of twelve

[6] Note that several other implementations of the product-scanning method, e.g. the one of Curve25519 [4], take advantage of Barrett's idea to increase performance.

Algorithm 2. Multiplication in $\mathbb{F}_p$

Input: Two 174-bit integers, $A = (a_5, a_4, a_3, a_2, a_1, a_0)$ and $B = (b_5, b_4, b_3, b_2, b_1, b_0)$, represented by six 29-bit words.
Output: Modular product $Z = A \cdot B \bmod (2^{174} - 3) = (z_5, z_4, z_3, z_2, z_1, z_0)$.

1: $s \leftarrow 0$
2: **for** i from 0 by 1 to 5 **do**
3: **for** j from 0 by 1 to i **do**
4: $s \leftarrow s + a_j \cdot b_{i-j}$
5: **end for**
6: $z_i \leftarrow s$ & 0x1fffffff
7: $s \leftarrow s \gg 29$
8: **end for**
9: **for** i from 6 by 1 to 10 **do**
10: **for** j from $i - 5$ by 1 to 5 **do**
11: $s \leftarrow s + a_j \cdot b_{i-j}$
12: **end for**
13: $z_i \leftarrow s$ & 0x1fffffff
14: $s \leftarrow s \gg 29$
15: **end for**
16: $z_{11} \leftarrow s$
17: $s \leftarrow 0$
18: **for** i from 0 by 1 to 5 **do**
19: $s \leftarrow z_i + 3 \cdot z_{i+6} + (s \gg 29)$
20: $z_i \leftarrow s$ & 0x1fffffff
21: **end for**
22: $z_0 \leftarrow z_0 + 3 \cdot (s \gg 29)$
23: **return** $Z = (z_5, z_4, z_3, z_2, z_1, z_0)$

29-bit words, i.e. $Z = (z_{11}, \ldots, z_1, z_0)$. This product has to be reduced modulo $p = 2^{174} - 3$ to obtain a final result within the range of $[0, p-1]$. Since p is a PM prime [9] of the form $2^k - c$, we can exploit the relation $2^k \equiv c \bmod p$ to accomplish the modular reduction in an efficient way. The reduction operation requires to split the product Z into a lower half Z_L (comprising, in our case, the six words $z_0, z_1, \ldots, z_5$) and an upper half Z_H (consisting of $z_6, z_7, \ldots, z_{11}$) so that $Z = Z_H \cdot 2^k + Z_L$. Now, Z can be reduced modulo $p = 2^k - c$ by simply substituting 2^k by c as shown in Equation (2).

$$Z \bmod p = Z_H \cdot 2^k + Z_L \bmod p = Z_H \cdot c + Z_L \bmod p \tag{2}$$

In essence, this boils down to multiplying Z_H by c and adding the product to Z_L, which leads to a result that is at most $(c + 1)$ times larger than p (i.e. the result is just a few bits longer than p if c is small). To get a completely reduced result, one can either perform the same procedure again, or simply subtract the prime p until a final result within the range of $[0, p-1]$ is obtained. The second part of Algorithm 2 (i.e. line 17 to 22) formally describes the reduction of the 348-bit product $Z = (z_{11}, \ldots, z_1, z_0)$ modulo $p = 2^{174} - 3$. Note that, in our implementation, the multiplication by 3 in line 19 is realized via three additions

of z_{i+6}. Therefore, reducing Z modulo $2^{174} - 3$ costs, in practice, no more than adding three 174-bit integers. The for-loop in line 18 – 21 is very similar to the loop of Algorithm 1 and, thus, can be implemented in Java in the same way as detailed earlier in this subsection for the addition in $\mathbb{F}_p$. Due to the 29-bit-per-word representation, it is possible to use a 32-bit variable of type `int` for the sum s and efficiently perform the additions in line 19 without overflow or loss of precision. The final result of Algorithm 2 may be not fully reduced and the least significant word z_0 may be up to 30 bits long instead of 29, similar to the result of Algorithm 1. However, as already mentioned, these "peculiarities" do not require special consideration since all arithmetic operations (bar inversion) are implemented such that they tolerate incompletely-reduced input operands and overlength (i.e. 30-bit) words. Note that Algorithm 2 is a simplified version of our actual Java implementation of multiplication in $\mathbb{F}_p$; we unrolled all loops to maximize performance. Furthermore, as stated in Subsection 2.1, we represent the field elements (i.e. 174-bit integers) by six individual variables of type `int` instead of arrays. These optimizations allow us to perform a multiplication without executing data-dependent branches or load operations, which helps to prevent certain forms of side-channel attack.

The square A^2 of a long (i.e. multiple-precision) integer A can be calculated considerably faster than the product of two distinct integers. When $A = B$, all partial products of the form $a_j \cdot b_{i-j}$ with $j \neq i - j$ in Algorithm 2 are identical to the partial products $a_{i-j} \cdot b_j$, i.e. they appear twice. However, an optimized squaring algorithm calculates each of these partial products only once and then shifts it left in order to double it. Due to our 29-bit-per-word representation, a 58-bit partial product held in a 64-bit `long` variable can be efficiently shifted to the left by one bit position without overflow. Multiplying two 174-bit integers represented by six 29-bit words requires a total of 36 word-level multiplications (i.e. `lmul` instructions in Java), whereas the squaring of a 174-bit integer costs only 21 word-level multiplications. In theory, squaring is nearly twice as fast as multiplication, but the difference in execution time decreases when taking the modular reduction into account. Modular squaring is generally only about 20% faster than modular multiplication since the reduction operation always takes the same time, regardless of whether a square or a product is reduced.

Inversion. We implemented the inversion in $\mathbb{F}_p$ according to the binary version of the Euclidean algorithm (Algorithm 2.22 in [15]) and applied some low-level optimizations such as multi-bit shifting. Even though inversion is costly, it has only little impact on scalar multiplication when using projective coordinates.

3 Point Arithmetic on GLV Curves

This section is devoted to the efficient implementation of scalar multiplication on a GLV curve and structured in a similar way as Section 2. We first explain the rationale behind choosing the curve $y^2 = x^3 - 7$ over $\mathbb{F}_p$ and then elaborate on different algorithms for scalar multiplication on GLV curves.

3.1 Selection of Elliptic Curve

At Crypto 2001, Gallant et al [12] introduced special families of elliptic curves over $\mathbb{F}_p$ that possess an endomorphism of small norm and demonstrated how to exploit this endomorphism to speed up a scalar multiplication. These so-called GLV curves can be seen as somewhat related to Koblitz curves over $\mathbb{F}_{2^m}$ since both provide a "shortcut" in the form of an endomorphism that allows one to perform a scalar multiplication significantly faster compared to random curves [15]. Gallant et al considered in [12] curves defined by the Weierstraß equation $y^2 = x^3 + ax$ (i.e. $b = 0$) and $y^2 = x^3 + b$ (i.e. $a = 0$) over a prime field $\mathbb{F}_p$ satisfying $p \equiv 1 \bmod 4$ in the former case and $p \equiv 1 \bmod 3$ in the latter. Both types of curve feature an endomorphism ϕ whose characteristic polynomial has small coefficients; in the former case the characteristic polynomial is $\lambda^2 + 1$ and in the latter case it is $\lambda^2 + \lambda + 1$ (see [15] for details). During the past ten years, the work of Gallant et al received considerable attention and the GLV method has been extended to hyperelliptic curves and recently to curves over $\mathbb{F}_{p^2}$ that are twists of curves defined over $\mathbb{F}_p$ (the so-called GLS curves [11]). However, we decided to use a Weierstraß curve of the form $y^2 = x^3 + b$ over $\mathbb{F}_p$ for our Java implementation since this choice of curve and field is more compliant with the major standards for EC cryptography (e.g. IEEE P1363 [17], SECG [24]) than hyperelliptic curves or elliptic curves over $\mathbb{F}_{p^2}$.

GLV curves are attractive to implementers for two reasons. First, when exploiting the endomorphism as per [12], scalar multiplication on GLV curves is considerably faster than on random curves, even though the speed-up is not as dramatic as for Koblitz curves [18]. Second, the GLV method can be applied in settings where arbitrary points are used as input for the scalar multiplication (e.g. in ECDH key exchange), i.e. the GLV method does not rely on a fixed base point that is known a priori. In the next subsection, we will briefly describe the GLV algorithm for scalar multiplication on curves of the form $y^2 = x^3 + b$. As mentioned above, these curves possess an efficiently computable endomorphism ϕ with characteristic polynomial $\lambda^2 + \lambda + 1$. Thanks to this endomorphism, it is possible to calculate the scalar multiplication $k \cdot P$ as $k_1 \cdot P + k_2 \cdot \phi(P)$, whereby k_1 and k_2 have only about half of the bitlength of the original k. These two half-length scalar multiplications can be carried out in an "interleaved" fashion using Shamir's trick [15], which halves the number of point doublings and also reduces the number of point additions compared to a conventional calculation of $k \cdot P$ using the double-and-add algorithm.

Finding a GLV curve with "good" cryptographic properties is a non-trivial task since, as mentioned in [5, Section 15.2], "the class of elliptic curves with an endomorphism of small norm is small." This task becomes harder still when one tries to find a combination of PM prime field and GLV curve so that both the field and the curve (resp. group) arithmetic can be implemented efficiently. We used the computer algebra package Magma to enumerate all PM primes of the form $2^k - c$ for $160 \leq k \leq 176$ and $c < 16$, and then search for each of them a GLV curve containing a large cyclic subgroup. Among the pairs of PM prime and GLV curve we found was $p = 2^{174} - 3$ and the curve defined through the

Weierstraß equation $y^2 = x^3 - 7$ (i.e. $a = 0$ and $b = p - 7$) over $\mathbb{F}_p$, which we finally decided to use for our Java implementation. Since a detailed description of the curve-finding methodology would go beyond the scope of this paper, we restrict ourselves to show that this specific GLV curve is suitable for use in EC cryptography. Both the IEEE standard P1363 [17] and the book of Cohen et al [5] serve as a good reference for security criteria that an elliptic curve has to fulfill; the most important ones are the following.

- First and foremost, the group of points on the curve has to contain a large subgroup of prime order n. In other words, the curve should have a small co-factor h; ideally, the co-factor is 1. Our curve $E : y^2 = x^3 - 7$ over the field $\mathbb{F}_p$ with $p = 2^{174} - 3$ has the order

  $$\#E(\mathbb{F}_p) = 23945242826029513411849172123055713727388153314904213,$$

  which happens to be a 174-bit prime; thus, the co-factor $h = \frac{\#E(\mathbb{F}_p)}{n}$ of this curve is 1. Following the notation from Section 1, a cryptosystem using the group $E(\mathbb{F}_p)$ provides a level of security that is well above the minimum (symmetric) level of 80 bits as recommended by ECRYPT II, the European Network of Excellence in Cryptology [22, Table 7.4].
- In order to circumvent the Semaev-Smart-Satoh-Araki (SSSA) attack, the curve must not be anomalous. This is obviously the case for our GLV curve since $\#E(\mathbb{F}_p) \neq p$.
- The embedding degree of the curve must not be small (i.e. the order of the EC group n must not divide $p^k - 1$ for "small" values of k) to prevent the Menezes-Okamoto-Vanstons (MOV) attack and other attacks based on the Weil and Tate pairing. Our GLV curve clearly satisfies this condition since $p^k \not\equiv 1 \bmod n$ for any $k \in [1, 100]$.

3.2 Efficient Implementation of Scalar Multiplication

A scalar multiplication is an operation of the form $k \cdot P$ whereby P denotes an EC point of large prime order n and k is an integer in the range $[1, n - 1]$ (see [15] for more information). Scalar multiplication in an EC group is nothing else than the repeated application of the group operation (i.e. point addition) on an element of the group, similar to exponentiation in a multiplicative group. It is common practice (at least when using curves over prime fields) to represent the points in projective coordinates as they allow for performing a point addition without inversion in the underlying finite field [15]. Our implementation of the point addition (resp. point doubling) is based on the mixed Jacobian-affine coordinates (resp. projective Jacobian coordinates) from Section 3.2.2 in [15]. An addition of points requires eight multiplications (8M), three squarings (3S), as well as a number of less-costly operations (e.g. additions, subtractions) in the underlying prime field, whereas a point doubling takes $4\mathrm{M} + 4\mathrm{S}$. However, the cost of point doubling on our GLV curve can be reduced by one multiplication to $3\mathrm{M} + 4\mathrm{S}$ since the parameter a is 0. Note that several variants of Jacobian

point addition/doubling formulae exist, some of which trade multiplications in $\mathbb{F}_p$ for squarings at the expense of an increased number of certain low-cost field operations[7]. However, none of these variants is capable to speed up our implementation due to the fact that the less-costly operations have a non-negligible execution time of between 0.1M and 0.27M (see Section 4). Thus, we decided to stick with the original formulae from [15] optimized for $a = 0$.

The most basic technique for performing a scalar multiplication $k \cdot P$ is the double-and-add method [15], which works in a similar way as the square-and-multiply method for exponentiation. Given a scalar k of a length of n bits, the double-and-add approach executes n point doublings and (in the average case) about $n/2$ point additions; the exact number of point additions depends on the Hamming weight of k. As mentioned above, a point addition on our GLV curve requires $8\mathrm{M} + 3\mathrm{S}$, whereas a point doubling takes only $3\mathrm{M} + 4\mathrm{S}$. Therefore, the overall cost of the double-and-add method amounts to $3n + 8n/2 = 7n$ multiplications and $4n + 3n/2 = 5.5n$ squarings in $\mathbb{F}_p$ (or, equivalently, $7\mathrm{M} + 5.5\mathrm{S}$ per bit of the scalar k). The average number of point additions can be reduced from $n/2$ to $n/3$ when the scalar k is represented in Non-Adjacent Form (NAF) [15]. In this case, the double-and-add method requires just $5.\dot{6}n$ multiplications and $5n$ squarings in the average case, i.e. $5.\dot{6}\mathrm{M} + 5\mathrm{S}$ per bit. However, a more significant reduction of execution time can be achieved by exploiting the endomorphism ϕ of our GLV curve as explained in [12]. This endomorphism makes it possible to obtain the scalar product $k \cdot P$ via $k_1 \cdot P + k_2 \cdot \phi(P)$, which is, in general, more efficient than a straightforward calculation of $k \cdot P$ since k_1 and k_2 have typically only half the bitlength of k and the two scalar multiplications $k_1 \cdot P$ and $k_2 \cdot \phi(P)$ can be carried out "simultaneously" (i.e. in an interleaved fashion) using Shamir's trick [15, Section 3.3.3].

In the following, we summarize some basic facts about our GLV curve and explain how to exploit its endomorphism ϕ for scalar multiplication, similar to Section 3.5 in [15]. Since the curve parameter a is 0 and the prime p satisfies $p \equiv 1 \bmod 3$, our GLV curve is of the type described in Example 4 in [12]. The underlying field $\mathbb{F}_p$ contains an element β of order 3 (because $p \equiv 1 \bmod 3$); in our implementation we use

$$\beta = 3940945791252506480086544614218081936071702723658 49$$

According to [12, Example 4], the map $\phi : E \to E$ defined by

$$\phi : (x, y) \mapsto (\beta x, y) \quad \text{and} \quad \phi : \mathcal{O} \mapsto \mathcal{O} \tag{3}$$

is an endomorphism of E defined over $\mathbb{F}_p$. The characteristic polynomial of ϕ is $\lambda^2 + \lambda + 1$. In order to exploit this endomorphism for scalar multiplication, we need a root modulo n of the characteristic polynomial, i.e. we need a solution to the equation $\lambda^2 + \lambda + 1 \equiv 0 \bmod n$; our implementation uses

$$\lambda = 7591969537352440260196679277338091599843085579298264$$

[7] Most of these variants can be found in the Explicit Formulas Database (EFD), an extensive repository of formulae for point arithmetic on various families of elliptic curves. The EFD is available online at `http://www.hyperelliptic.org/EFD`.

Table 1. Timings of the field arithmetic on different devices.

Operation	T60	HTC	X2	6610
Addition	24.2 ns	124.2 ns	1319 ns	63.1 μs
Subtraction	24.4 ns	130.0 ns	1772 ns	64.6 μs
Multiplication	255.5 ns	994.4 ns	7256 ns	232.3 μs
Squaring	197.9 ns	860.2 ns	6702 ns	186.9 μs
Inversion	26,6 μs	191.4 μs	1479 μs	54,8 ms

The solution λ has the property that $\phi(P) = \lambda \cdot P$ for all $P \in E(\mathbb{F}_p)$ [15]. Note that computing $\phi(P)$ for a point $P = (x, y)$ requires only one multiplication in $\mathbb{F}_p$, namely $\beta \cdot x$. As stated before, the common strategy for computing $k \cdot P$ on a GLV curve is to decompose the n-bit scalar k into two "half-length" integers k_1 and k_2 (often referred to as *balanced length-two representation* of k [15]) so that $k = k_1 + k_2\lambda \bmod n$. This decomposition of k into k_1 and k_2 is described in detail in [15] and requires merely a few multi-precision multiplications if the curve and field are fixed. As $k \cdot P = k_1 \cdot P + k_2 \cdot \lambda \cdot P = k_1 \cdot P + k_2 \cdot \phi(P)$, the result of $k \cdot P$ can be obtained by first computing $\phi(P)$ (which takes one single field multiplication) and then using a simultaneous double-scalar multiplication ("Shamir's trick") to perform these two half-length scalar multiplications in an interleaved fashion. Consequently, the GLV requires only $n/2$ point doublings for an n-bit scalar multiplication, which corresponds to a 50% reduction compared to the double-and-add method. The number of point additions depends on the joint Hamming density of k_1 and k_2; in the average case (i.e. when the joint Hamming density is 0.75), a total of $0.75 \cdot n/2 = 0.375n$ point additions must be carried out. In summary, the overall cost of the GLV method amounts to $0.5n \cdot 3 + 0.375n \cdot 8 = 4.3n$ multiplications and $0.5n \cdot 4 + 0.375n \cdot 3 = 3.125n$ squarings in $\mathbb{F}_p$, i.e. $4.3\mathrm{M} + 3.125\mathrm{S}$ per bit. However, the Hamming density can be reduced to 0.5 (on average) by representing k_1 and k_2 in Joint Sparse Form (JSF) [23], which, in turn, cuts the number of point additions by roughly one third to $0.5 \cdot n/2 = 0.25n$. In this case, the total cost of computing $k \cdot P$ on a GLV curve is reduced to, on average, $0.5n \cdot 3 + 0.25n \cdot 8 = 3.5n$ multiplications and $0.5n \cdot 4 + 0.25n \cdot 3 = 2.75n$ squarings in $\mathbb{F}_p$, i.e. $3.5\mathrm{M} + 2.75\mathrm{S}$ per bit.

4 Implementation Results and Discussion

We evaluated the performance (i.e. execution time) of our Java implementation of the field arithmetic and different algorithms for scalar multiplication on an IBM Thinkpad T60 and a set of mobile phones, including an HTC Desire S, a Nokia X2, and an old Nokia 6610. The most powerful of these test devices is the Thinkpad T60; it features a 32-bit Intel T2300 ("Core Duo") processor clocked with a frequency of 1.66 GHz. At the opposite end of the spectrum is the Nokia 6610, which is equipped with an ARM 11 processor clocked at 104 MHz. Table 1 summarizes the execution times of the field arithmetic operations on the T60 and the three mobile phones. Given the differing computational power of these

Table 2. Comparison of scalar multiplication methods on different devices

Algorithm	Cost per bit	T60	HTC	X2	6610
Dbl-and-Add	$7\mathrm{M} + 5.5\mathrm{S}$	569.1 μs	2745 μs	19.79 ms	632.5 ms
Dbl-and-Add (NAF)	$5.\dot{6}\mathrm{M} + 5\mathrm{S}$	490.3 μs	2352 μs	17.24 ms	552.9 ms
GLV method	$4.3\mathrm{M} + 3.125\mathrm{S}$	362.5 μs	2057 μs	13.36 ms	437.4 ms
GLV method (JSF)	$3.5\mathrm{M} + 2.75\mathrm{S}$	326.9 μs	1885 μs	12.20 ms	400.2 ms

devices, it is not surprising that the measured execution times vary by a large extent. However, besides the differences in processing speed, one must also take into account the capabilities of the particular JVM of each device. On the T60 we used the HotSpot Client VM that comes with Sun's JDK 6 Update 27; this VM features a JiT compiler for efficient execution of bytecodes. Most modern smart phones have either a VM with JiT compilation or a processor with Java extensions. On the other hand, the old Nokia 6610 provides a conventional VM without JiT compiler or any other form of Java acceleration. In order to obtain accurate timings, we put each arithmetic function into a loop and performed a sufficiently large number of iterations so that the overall execution time was in the range of several seconds. We measured the execution time with help of the `currentTimeMillis` function, which is provided by the `System` class. Table 1 specifies the average execution time (i.e. the quotient of overall execution time and number of iterations) of the arithmetic operations.

On the Thinkpad T60, the relative execution times of the arithmetic operations are very similar to that of previous implementations (e.g. [15]) written in C and/or Assembly language. Squaring is about 22% faster than multiplication (i.e. $\mathrm{S} = 0.78\mathrm{M}$), whereas addition and subtraction (and similar operations like halving or negation of a field element) execute in roughly 0.1M. The inversion is slow compared to the multiplication ($\mathrm{I} = 116\mathrm{M}$), but this not surprising since also most C (and Assembly) implementations of prime-field arithmetic have an I/M ratio of between 50 and 100 [15]. However, the relations are different on the Nokia 6610 because addition and subtraction take more than one fourth of the execution time of a multiplication; the exact ratio of addition to multiplication is 0.27. Also the inversion is very slow compared to the multiplication; based on the results from Table 1 we have $\mathrm{I} = 235\mathrm{M}$. An explanation for the differences between the T60 and the Nokia 6610 can be found in the characteristics of the underlying processor. The Core Duo of the T60 is a superscalar processor able to execute several arithmetic/logical instructions in parallel, which has a positive effect on the execution time of addition, subtraction, and inversion. On the other hand, the Core Duo can execute only a single integer multiply instruction at a time, i.e. multiplication and squaring in $\mathbb{F}_p$ can take little advantage of the superscalar pipeline. In contrast, the ARM 11 processor of the Nokia 6610 has a single-issue pipeline with a relatively fast integer multiplier, which favors the field multiplication over addition, subtraction, and inversion.

Table 2 shows a comparison of the four algorithms for scalar multiplication discussed in Subsection 3.2. The actual execution times on both the Thinkpad

T60 and the mobile phones roughly match with the cost-per-bit figures given in the second column when taking into account that this approximate cost model only considers multiplications and squarings in $\mathbb{F}_p$. More concretely, on the T60 the GLV method with JSF representation of k_1, k_2 is 1.74 times faster than the standard double-and-add technique, which is relatively close to the theoretical speed-up factor of 2.0 suggested by the approximate cost model. However, as stated before, the approximate cost model is based on the number of multiplications and squarings in $\mathbb{F}_p$ and ignores that the GLV method has to perform other operations such as the decomposition of k into k_1, k_2 and the calculation of the sum $P + \phi(P)$ in affine coordinates, which may involve a costly inversion in $\mathbb{F}_p$. The T60 executes the GLV method in merely 326.9 μs; this translates to over 3,000 scalar multiplications per second. In comparison, the Bouncy Castle Lightweight Crypto API [26] requires 14.95 ms for a 174-bit scalar multiplication, i.e. our implementation outperforms Bouncy Castle by a factor of 45. On the 6610, the GLV method is only 1.58 times faster than the double-and-add method, mainly because of the fact that addition/subtraction and inversion are relatively more costly (versus multiplication) than on the T60. Nonetheless, the Nokia 6610 is able to execute the GLV method in about 400 ms, which is 5.38 times faster than the Java implementation of EC scalar multiplication over the binary extension field $\mathbb{F}_{2^{191}}$ reported in [27].

The GLV method compares very well with other approaches for scalar multiplication and other forms of elliptic curves, e.g. Montgomery [21] or Edwards curves [10]. A Montgomery curve over $\mathbb{F}_p$ is defined by an equation of the form $By^2 = x^3 + Ax^2 + x$ with $(A^2 - 4)B \neq 0$ and allows for fast computation of the x-coordinate of the sum $P + Q$ of two points P, Q whose difference $P - Q$ is known. More precisely, a point addition performed according to the formula in [21, p. 261] requires four multiplications (4M) and two squarings (2S), whereas a point doubling costs 3M and 2S. However, one of the three multiplications in the point doubling uses the constant $(A - 2)/4$ as operand, which is small if the parameter A is chosen properly. Our experiments show that multiplying a field element by a small (up to 29 bits) constant costs some 0.2M. Furthermore, the point addition formula given in [21, page 261] can be optimized when using the so-called Montgomery ladder (Algorithm 13.35 in [5]) for scalar multiplication and representing the base point in projective coordinates with $Z = 1$ (see also Remark 13.36 (ii) in [5]). In this case, a "ladder"-based implementation of the scalar multiplication requires exactly $5.2n$ multiplications and $4n$ squarings in $\mathbb{F}_p$, i.e. 5.2M + 4S per bit, which is still a lot more than the 3.5M + 2.75S of the GLV method with JSF-representation of the two half-length scalars.

5 Conclusions

We introduced a Java implementation of scalar multiplication on a GLV curve over a 174-bit prime field. Our implementation supports arbitrary base points (making it suitable for ECDH key exchange) and is optimized to reach high performance, low memory footprint and small bytecode size. Furthermore, the

described implementation is highly self-contained as it needs only a few classes from the standard Java class library. On a Thinkpad T60 with a 1.66 GHz Core Duo processor, our GLV method reaches a throughput of 3,000 scalar multiplications per second, which is about 45 times higher than the throughput of the widely-used Bouncy Castle library for J2ME. The scalar multiplication time on mobile phones ranges from less than 2 ms (HTC Desire S) to roughly 400.6 ms (Nokia 6610). We attribute the high performance of our GLV technique to the efficiency of both the field and group arithmetic. The radix-2^{29} representation of field elements allows for a branch-less implementation of the field arithmetic modulo $p = 2^{174} - 3$, which facilitates the execution on JiT-enabled JVMs. We also demonstrated that, when exploiting their endomorphism, GLV curves are considerably faster than ordinary Weierstraß curves or Montgomery curves. In summary, our measured results confirm the great potential of GLV curves and PM prime fields for the implementation of high-speed EC cryptography.

The full Java source code of the implementation described in this paper is available for download at `https://cryptolux.org`.

References

1. ARM Limited: Jazelle^{TM} – ARM® Architecture Extensions for Java Applications. White paper, available for download at `http://www.arm.com/armtech/jazelle?OpenDocument` (Oct 2000)
2. Arnold, K., Gosling, J., Holmes, D.: The Java^{TM} Programming Language. Prentice Hall, fourth edn. (2005)
3. Barrett, P.D.: Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) Advances in Cryptology — CRYPTO '86. Lecture Notes in Computer Science, vol. 263, pp. 311–323. Springer Verlag (1987)
4. Bernstein, D.J.: Curve25519: New Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) Public Key Cryptography — PKC 2006. Lecture Notes in Computer Science, vol. 3958, pp. 207–228. Springer Verlag (2006)
5. Cohen, H., Frey, G.: Handbook of Elliptic and Hyperelliptic Curve Cryptography, Discrete Mathematics and Its Applications, vol. 34. Chapmann & Hall\CRC (2006)
6. Comba, P.G.: Exponentiation cryptosystems on the IBM PC. IBM Systems Journal 29(4), 526–538 (Dec 1990)
7. Craig, I.D.: Virtual Machines. Springer Verlag (2006)
8. Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., Wolczko, M.: Compiling Java just in time. IEEE Micro 17(3), 36–43 (May/June 1997)
9. Crandall, R.E.: Method and apparatus for public key exchange in a cryptographic system (Oct 1992), U.S. Patent No. 5,159,632
10. Edwards, H.M.: A normal form for elliptic curves. Bulletin of the American Mathematical Society 44(3), 393–422 (Jul 2007)
11. Galbraith, S.D., Lin, X., Scott, M.: Endomorphisms for faster elliptic curve cryptography on a large class of curves. In: Joux, A. (ed.) Advances in Cryptology — EUROCRYPT 2009. Lecture Notes in Computer Science, vol. 5479, pp. 518–535. Springer Verlag (2009)
12. Gallant, R.P., Lambert, R.J., Vanstone, S.A.: Faster point multiplication on elliptic curves with efficient endomorphism. In: Kilian, J. (ed.) Advances in Cryptology —

CRYPTO 2001. Lecture Notes in Computer Science, vol. 2139, pp. 190–200. Springer Verlag (2001)
13. Gosling, J., McGilton, H.: The Java™ Language Environment. White paper, Sun Microsystems, Inc., Mountain View, CA, USA (May 1996)
14. Großschädl, J., Avanzi, R.M., Savaş, E., Tillich, S.: Energy-efficient software implementation of long integer modular arithmetic. In: Rao, J.R., Sunar, B. (eds.) Cryptographic Hardware and Embedded Systems — CHES 2005. Lecture Notes in Computer Science, vol. 3659, pp. 75–90. Springer Verlag (2005)
15. Hankerson, D.R., Menezes, A.J., Vanstone, S.A.: Guide to Elliptic Curve Cryptography. Springer Verlag (2004)
16. Hennessy, J.L., Jouppi, N.P., Baskett, F., Gross, T.R., Gill, J.: Hardware/software tradeoffs for increased performance. In: Proceedings of the 1st International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1982). pp. 2–11. ACM Press (1982)
17. Institute of Electrical and Electronics Engineers (IEEE): IEEE Std 1363-2000: IEEE Standard Specifications for Public-Key Cryptography (Aug 2000)
18. Koblitz, N.I.: CM-curves with good cryptographic properties. In: Feigenbaum, J. (ed.) Advances in Cryptology — CRYPTO '91. Lecture Notes in Computer Science, vol. 576, pp. 279–287. Springer Verlag (1992)
19. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press (1996)
20. Möller, B.: Securing elliptic curve point multiplication against side-channel attacks. In: Davida, G.I., Frankel, Y. (eds.) Information Security — ISC 2001. Lecture Notes in Computer Science, vol. 2200, pp. 324–334. Springer Verlag (2001)
21. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48(177), 243–264 (Jan 1987)
22. Smart, N.P. (ed.): ECRYPT II Yearly Report on Algorithms and Keysizes (2009-2010). European Network of Excellence in Cryptology (ECRYPT II) (Mar 2010), deliverable D.SPA.13, available for download at `http://www.ecrypt.eu.org/documents/D.SPA.13.pdf`
23. Solinas, J.A.: Low-weight binary representations for pairs of integers. Tech. Rep. CORR 2001-41, Centre for Applied Cryptographic Research (CACR), University of Waterloo, Waterloo, Canada (2001)
24. Standards for Efficient Cryptography Group (SECG): SEC 1: Elliptic Curve Cryptography. Available for download at `http://www.secg.org` (May 2009)
25. Stiftung Secure Information and Communication Technologies (SIC): IAIK JCE Micro Edition (Version 3.04). Available for download at `http://jce.iaik.tugraz.at/sic/Products/Mobile-Security/JCE-ME` (Sep 2006)
26. The Legion of the Bouncy Castle: Lightweight Cryptography API (Release 1.47). Available for download at `http://www.bouncycastle.org` (Mar 2012)
27. Tillich, S., Großschädl, J.: A survey of public-key cryptography on J2ME-enabled mobile devices. In: Aykanat, C., Dayar, T., Korpeoglu, I. (eds.) Computer and Information Sciences — ISCIS 2004. Lecture Notes in Computer Science, vol. 3280, pp. 935–944. Springer Verlag (2004)
28. White, J.P., Hemphill, D.A.: Java 2 Micro Edition. Manning Publications (2002)
29. Wiener, M.J., Zuccherato, R.J.: Faster attacks on elliptic curve cryptosystems. In: Tavares, S.E., Meijer, H. (eds.) Selected Areas in Cryptography — SAC '98. Lecture Notes in Computer Science, vol. 1556, pp. 190–200. Springer Verlag (1999)
30. Yanık, T., Savaş, E., Koç, Ç.K.: Incomplete reduction in modular arithmetic. IEE Proceedings – Computers and Digital Techniques 149(2), 46–52 (Mar 2002)