Batch Binary Weierstrass

Billy Bob Brumley, Sohaib ul Hassan, Alex Shaindlin, Nicola Tuveri, and Kide Vuojärvi

Tampere University, Tampere, Finland
{billy.brumley,n.sohaibulhassan,chloenatasha.shaindlin}@tuni.fi
{nicola.tuveri,kide.vuojarvi}@tuni.fi

Abstract. Bitslicing is a programming technique that offers several attractive features, such as timing attack resistance, high amortized performance in batch computation, and architecture independence. On the symmetric crypto side, this technique sees wide real-world deployment, in particular for block ciphers with naturally parallel modes. However, the asymmetric side lags in application, seemingly due to the rigidity of the batch computation requirement. In this paper, we build on existing bitsliced binary field arithmetic results to develop a tool that optimizes performance of binary fields at any size on a given architecture. We then provide an ECC layer, with support for arbitrary binary curves. Finally, we integrate into our novel dynamic OpenSSL engine, transparently exposing the batch results to the OpenSSL library and linking applications to achieve significant performance and security gains for key pair generation, ECDSA signing, and (half of) ECDH across a wide range of curves, both standardized and non-standard.

Keywords: applied cryptography · public key cryptography · elliptic curve cryptography · software implementation · batching · bitslicing · OpenSSL

1 Introduction

The use of Elliptic Curve Cryptography (ECC) was first suggested in 1985, independently by Miller [23] and Koblitz [19]. Due to the fast group law on elliptic curves and the absence of known sub-exponential attacks, ECC has since gathered momentum as an alternative to popular asymmetric cryptosystems like RSA and DSA, as it can provide the same security level with keys that are much shorter, gaining both in computational efficiency and bandwidth consumption.

A prevalent assumption is that for software implementations, curves over large characteristic prime fields are generally more efficient than their binary field counterparts, and that, vice versa, the opposite applies for hardware implementations, i.e., that ECC scalar point multiplication (usually the most costly operation in ECC cryptosystems) is much faster over binary extension fields.

As we will detail in the following sections, the first half of the above assumption has been repeatedly challenged, especially considering the effect of alternative coordinate representations and the advances in the manufacturing processes

and design of general purpose CPUs, e.g. with the introduction of dedicated units for carry-less multiplication and the widespread adoption of vector processing (a.k.a. SIMD: Single Instruction, Multiple Data) in popular Instruction Set Architectures (ISA).

In this work, we focus on bitsliced binary field arithmetic as a strategy to take advantage of the characteristics of modern processors and provide fast batch "fixed point"[1] scalar multiplication for binary ECC.

Our Contribution. We propose a set of three tools

1. to optimize bitsliced binary field arithmetic, potentially supporting any architecture, by selecting the performance-optimal configuration of the underlying finite field layer for a given platform;
2. to implement an ECC layer for any given binary curve, building on the layer provided by the first tool, and providing constant-time fixed point scalar multiplications that performs very competitively with existing state-of-the-art results;
3. to integrate the generated ECC implementations in OpenSSL, overcoming the restriction of batch computation at the application level.

The combined output of these tools transparently provides constant-time and efficient implementations of bitsliced binary ECC for real-world applications built on top of OpenSSL. This challenges once again the notion that binary ECC are not well suited for software implementations, and at the same time overcomes the main drawback of similar techniques, proving it is also practical for real-world deployment.

Overview. In Section 2 we discuss the background for this work, recalling related works on top of which we build our contribution or that pursued similar goals. In Section 3 and Section 4 we discuss definitions, challenges, the design, and the analysis of our results related with the binary field and elliptic curve arithmetic layers. Section 5 describes the third and last of our contributions, integrating our implementations in OpenSSL to provide seamless support of constant-time and fast bitsliced binary ECC to real-world applications. Finally, we conclude in Section 6, with a discussion of the limits of our current contribution and future work directions for this research.

2 Background

2.1 Bitslicing

SIMD is a data parallelism technique that facilitates parallel computation on multiple values. For example, processors featuring AVX2 contain 256-bit registers

[1] In this paper we refer to scalar multiplication by the conventional generator point for a given curve as *"fixed point" scalar multiplication*. In real world applications, this is the fundamental operation for ECDH and ECDSA key generation, and for ECDSA signature generation. This operation is opposed to *"generic point" scalar multiplication*, intended as a scalar multiplication by any other point on the curve: the latter is used in ECDH key derivation and ECDSA signature verification.

`ymm0` through `ymm15`. Viewing these as four 64-bit *lanes*, the instruction `vpaddq %ymm0, %ymm1, %ymm2` takes `ymm0` $= (a_0, a_1, a_2, a_3)$ and `ymm1` $= (b_0, b_1, b_2, b_3)$ then produces their integer vector sum `ymm2` $= (a_0 + b_0, \ldots, a_3 + b_3)$ where all sums are modulo 2^{64}. There are several ways to split the register; e.g. as 8-bit, 16-bit, etc. lanes, each requiring dedicated microarchitecture support and distinct instructions for the register-size variants (e.g. add, subtract, multiply, etc.). Put briefly, *bitslicing* takes this to the extreme and views any w-bit register as w 1-bit lanes. This lightens the microarchitecture requirements, as 1-bit addition (or subtraction) is simply bitwise-XOR; 1-bit multiplication is bitwise-AND. Bitwise operations are a fundamental feature in ISAs, and are in fact independent of explicit SIMD support: any generic, non-SIMD w-bit architecture natively supports 1-bit lane SIMD, i.e. bitslicing.

Deployments. Outside of primitives that integrate bitslicing into the design such as Serpent, SHA-3, and Ascon, we are aware of two large-scale deployments of bitsliced software: one defensive and the other offensive. Käsper and Schwabe [18] provide a bitsliced implementation for AES, mainlined by OpenSSL in 2011. In the pre-AES-NI era and with only SSE2 as a prerequisite, this was groundbreaking work that exceeded the performance of traditional table-based AES software, yet additionally provided timing attack resistance. *John the Ripper*[2] is a security audit tool that bitslices DES to batch password hashing. The main application is to hashes utilizing the `crypt` portion of the standard library.

Obstacles. The ability to batch public key operations is the largest restriction for bitsliced software. Real-world APIs (e.g. OpenSSL) do not support such a seemingly narrow use case, as e.g. in ECC the most common operations are single or double scalar multiplications, not w in parallel. Even research-oriented APIs such as SUPERCOP do not have this feature. And for key agreement on a typical single threaded application, there is no clear way how to utilize such an implementation. On the engineering side, it is very tempting to directly leverage academic results on minimizing gate count for bitsliced software, since each gate will map to an instruction. However, this is only part of the story since register and memory pressure impose constraints, as well as instruction level parallelism, scheduling, and binary size. Indeed, both [8, 34] note the latter disconnect. Binary finite field multipliers with low gate count [5] do not directly map to efficient bitsliced multipliers, since arithmetic instruction count is only a small part of overall performance on a platform.

2.2 OpenSSL and `ENGINE`s

As already mentioned, arguably the main drawback of batch operations for cryptographic implementations is that they are generally seen as not practical: this comes in part by the lack of support for batch public key operations in mainstream cryptographic libraries. Among many others, one example supporting this argument is `BBE251` by Bernstein [5]: despite the high performance, we are

[2] https://www.openwall.com/john/

not aware of any deployments or standardization efforts supporting this curve in the past decade. We hypothesize this is because it seemingly does not meet the characteristics for mainstream cryptography software.

To counter this argument and to evaluate our work in a real-world scenario, we decided to target OpenSSL, an open source project consisting of a general-purpose cryptographic library, an SSL/TLS library and toolkit, and a collection of command line tools to generate and handle cryptographic keys and execute cryptographic operations. The project is arguably ubiquitous, providing the cryptographic backend and TLS stack of a considerable portion of web servers, network appliances, client softwares, and IoT devices. Thanks to the wide range of supported platforms and more than twenty years of history, it has become a de facto standard for Internet Security.

In the literature, a common pattern to integrate alternative implementations in OpenSSL consists in forking the upstream project to apply the required patches. This then requires maintaining the fork to include, alongside the research-driven changes, patches from upstream to fix vulnerabilities, bugs, or provide new features. As an example of this methodology, the Open Quantum Safe project [30] maintains a fork[3] of OpenSSL to evaluate candidates of the NIST Post Quantum Project. The history of the repository shows the level of effort required to maintain a fork of OpenSSL up to date with both research work and upstream releases.

Considering how demanding this approach can be, it is not surprising that most of the academic results often prefer to evaluate their results with ad-hoc software or toolkits like SUPERCOP[4], which generally operate in isolation and are not necessarily representative of the performance or features of the applications of the research work in real-world use cases.

As an alternative to these two approaches, we build on top of the framework proposed by Tuveri and Brumley [33], instantiated in `libsuola`[5]. This framework allows to provide alternative implementations of cryptosystems to OpenSSL applications, by using `ENGINE`s: OpenSSL objects that act as "containers for implementations of cryptographic algorithms". Originally introduced to support hardware cryptographic accelerators, the same construct can be used to provide alternative software implementations. It offers a mechanism to configure a whole system or individual applications to load `ENGINE`s at runtime, transparently providing their functionality to existing applications, without recompiling them.

While we defer to Section 5 for a description of the `ENGINE` instantiated as part of our contribution, we further motivate here our choice remarking that this approach, on top of lowering maintenance costs, allows to reuse existing applications with no effort, providing multiple and diverse ways to validate the correctness and interoperability of our implementation. This also allows us to

[3] https://github.com/open-quantum-safe/openssl
[4] https://bench.cr.yp.to/supercop.html
[5] https://github.com/romen/libsuola

use existing projects to benchmark and evaluate our contribution in comparison with state-of-the-art real-world implementations.

3 Binary Field Arithmetic

A binary extension field is a finite field of the form $\mathbb{F}_{2^m}$, where m is an integer called the dimension of the field and $m \geq 2$. Elements of the field $\mathbb{F}_{2^m}$ can be expressed as polynomials of degree less than m with coefficients in $\mathbb{F}_2$, which have an underlying set of $\{0, 1\}$ and in which addition corresponds to binary `XOR` and multiplication corresponds to binary `AND`. Any two finite fields with the same number of elements are isomorphic to each other, but calculations in a particular finite field are performed modulo an irreducible polynomial P of degree m, and different choices of P produce different results. Since the underlying set of $\mathbb{F}_2$ is $\{0, 1\}$, elements of $\mathbb{F}_{2^m}$ can also be represented as binary strings; for example, with $m = 8$, the element $x^6 + x^3 + x + 1$ can be written 01001011_2.

The simplest method of multiplying elements of $\mathbb{F}_{2^m}$ is to multiply them as polynomials using schoolbook multiplication and then reduce the result modulo the field polynomial P by polynomial long division. Bernstein [5, Sect. 2] collects many asymptotic improvements on $M(n)$, the number of bit operations required to multiply two n-bit polynomials, over this method: $M(n) \leq \Theta(n^{\lg 3})$ due to Karatsuba, $M(n) \leq n2^{\Theta(\sqrt{\lg n})}$ due to Toom, and $M(n) \leq \Theta(n \lg n \lg \lg n)$ due to Schönhage and Strassen. Bernstein also establishes tighter explicit upper bounds on $M(n)$ for $n \in \{128, 163, 193, 194, 512\}$, and provides[6] straight-line code for cases from $n = 1$ to $n = 1000$, and verified upper bounds on the number of bit operations required in each case.

3.1 Splitting Strategies

For small n, the straight-line code can be very efficient, but for large n, it becomes inefficient, partly because the compiled code becomes too large to fit in the cache. Additionally, the algorithm that uses the fewest bit operations will not necessarily take the fewest cycles to run when implemented, because in bitsliced batch computations a nontrivial number of cycles are spent performing load and store instructions, which is not accounted for in the bit operation count. There are several conventional concerns about the correlation of bit operation count and software performance, and while bitslicing relieves some of these concerns [5, Sec. 1], the overhead incurred by loads and stores remains relevant. This is a gap in the existing literature, which mostly reports results in terms of bit operations [11, 10, 16].

Recursive algorithms for polynomial multiplication, such as Karatsuba and Toom, have better asymptotic performance than straight-line multiplication, but they incur more overhead, so for sufficiently small inputs, straight-line multiplication is faster in practice. Due to these considerations, the fastest way (in terms

[6] https://binary.cr.yp.to/m.html

of cycle count) to batch multiply polynomials of cryptographic sizes in $\mathbb{F}_{2^m}$ is usually to begin with recursive splits, and then switch to straight-line multiplication when the subproblem size becomes small enough for this trade-off to occur. The exact size at which this threshold is located may depend on both the architecture and the field dimension.

For larger subproblems, there are many different recursive algorithms for polynomial multiplication. Of the recursive multiplication strategies described in [5], we use two: the one called "five-way recursion", which corresponds to the `WAY3` macros in our implementation, and the one called "two-level seven-way recursion", which corresponds to the `WAY4` macros. (There is also "three-way recursion", corresponding to `WAY2` macros, but we omit them; since `WAY4` splits the current problem into four subproblems of roughly equal size, and `WAY2` splits it into two subproblems of roughly equal size, we expect that `WAY4` is a more efficient version of back-to-back `WAY2` splits.) For simplicity, and in keeping with the names of the macros, in this paper we refer to the `WAY3` macros as "three-way recursion" and to the `WAY4` macros as "four-way recursion". Different subproblem sizes may have different optimal choices of recursion strategy (and again, these can be architecture-dependent), so the complete collection of recursive multiplication steps taken before dropping down to straight-line multiplication may be both architecture- and dimension-dependent.

For a given field size, on a given architecture, a strategy to choose whether to use a recursive algorithm or switch to a straight-line multiplication at each intermediate step needs to be created. Said strategy aims to minimize the total number of cycles required to multiply a batch of w polynomials of the given degree. We refer to this result as the optimal *splitting strategy* for that size. The strategy is generated for the library and necessary straight-line multiplication files are included.

The functions for straight-line multiplication are called `gf2_mul_M` and the functions for recursive multiplication are called `karatmultM`, where `M` is the size of the input. Reading Figure 1 from the bottom line up, this code splits 251 four ways with a `WAY4` macro (specifically `WAY43` because $251 \equiv 3 \mod 4$) and performs recursive multiplication on subproblems of size 62 and 63; 63 is split four ways with a remainder of 3 and 62 is split four ways with a remainder of 2; both 62 and 63 are split into subproblems of size 15 and 16, which are handled with straight-line code.

```
/* (43K251, 43K63, 42K62, G16, G15) */
WAY42(62, gf2_mul_15, gf2_mul_16)
WAY43(63, gf2_mul_15, gf2_mul_16)
WAY43(251, karatmult62, karatmult63)
```

Fig. 1. Optimal splitting strategy on Skylake for $m = 251$.

When reporting benchmarking results, we display the splitting strategy in a concise format: a list of multipliers in descending order of subproblem size, with recursive multipliers represented by the numbers after the `WAY` macro + `K` + the input size, and straight-line multipliers represented by `G` + the input size.

Architectures. The purpose of the benchmarking tool we developed is to experimentally determine the best splitting strategy for a particular field size running on a particular architecture. The tooling currently supports AVX2, AVX-512, and NEON, but is easily extendable to other ISAs. The bulk of our code utilizes macros for C compiler intrinsics to emit architecture-specific instructions, so adding an ISA consists mainly of internally defining these macros for the target architecture. We used the following environments for the benchmarking: *AVX-512*, a 2.1GHz Xeon Silver 4116 Skylake (24 cores, 48 threads across 2 CPUs) and 256GB RAM running 64-bit Ubuntu 16.04 Xenial (`clang-8`, $w = 512$); *AVX2*, a 3.2GHz i5-6500 Skylake (4 cores) and 16GB RAM running 64-bit Ubuntu 18.04 Bionic (`clang-8`, $w = 256$); *AVX2-AMD*, a 3.7GHz Ryzen 7 2700X (8 cores, 16 threads) and 16GB RAM running 64-bit Ubuntu 18.04 Bionic (`clang-8`, $w = 256$); *NEON*, a Raspberry Pi 3 Model B+, 1.4GHz Broadcom BCM2837B0 ARMv8 Cortex-A53 (4 cores) and 1GB RAM running 64-bit Ubuntu 18.04 Bionic (`clang-7`, $w = 128$).

3.2 Benchmarking

The benchmarking for the new software was done by recursively generating different splitting strategies for the multiplication of the polynomials. The dimension of the original field is recursively split three-ways and four-ways until the limits for straight-line multiplication are reached. While within the limits, strategies are generated for both recursive and straight-line multiplication, because we found that using straight-line multiplication was not always the optimal solution even when they were available. Thus we also generate many strategies that still use the recursive split while within the limits of straight-line multiplication. While Bernstein [5] has straight-line multiplications defined in a larger range, we decided to limit it between polynomials of degree $[5, \ldots, 99]$ for the scope of this paper.

We divided the generated splitting strategies into three categories, two of which were eliminated from this paper. The two eliminated categories consisted of *mixed multiplication* and *non-strict recursion threshold*. The nature of these categories and the reasons for elimination is discussed below.

The *mixed multiplication* category includes all the strategies where at least one subproblem of the recursive call is not handled like the others. An example of this would be `WAY43(251, karatmult62, gf2_mul_63)`, where one subproblem is handled with a recursive call, and another with straight-line multiplication. The benchmarking tool still supports this option, but our preliminary testing showed no benefits for allowing mixed multiplication. The number of possible strategies is also vastly larger (6 vs. 11 with degree 63, 14 vs. 62 with degree 127 and 193 vs. 4546 with degree 251), rendering the tool prohibitively slow with

higher values. Though there may be some edge cases where using a recursive call for the higher value and straight-line for the lower value subproblem would yield a more optimal result, we found none.

The *non-strict recursion threshold* category includes all the strategies where the greatest value of straight-line multiplication is greater or equal to the least value of a recursive multiplication. An example of this would be the strategy `(43K251, 30K63, 32K62, G23, 42K22, 41K21, G21, 40K20, G6, G5)`, where we see both a G23 which has a greater value than K22, and K21 which is equal in value to G21. Our assumption is that once we cross the threshold on straight-line multiplication, using recursive multiplication will be slower. As is the case with mixed multiplication, limiting the search space by excluding this category considerably increases the efficiency of the benchmarking tool. We believe that this elimination does not significantly reduce the chances of finding the optimal strategy. If straight-line and recursive multiplication of the same subproblem size, such as G21 and K21, is wanted inside one strategy, the current version of the benchmarking software needs to be modified.

Unlike the mixed multiplication strategies, the strategies with non-strict recursion threshold are only eliminated after all the strategies have been generated. The final search space includes all the strategies that did not meet the criteria for elimination in the previous two steps.

After the paths have been generated, the benchmarking tool takes one strategy at a time, creates a configuration header file for the binary field arithmetic C program, and compiles and runs the test harness, which outputs the number of processor cycles it takes for a batch of polynomials to be multiplied. When all the strategies for that field size have been tested, the program outputs a file containing the strategies in ascending order of cycles, as well as the configuration file for the best found strategy for a given platform. Our ECC layer (Section 4) uses this configuration at build time to produce the most efficient solution.

Figure 2 shows the results of the best strategy in benchmarking on three different processor architectures, and Table 1 selective data points on four different processor architectures. NEON has a register width of 128 bits, AVX2 has a register width of 256 bits, and AVX-512 has a register width of 512 bits; these are the denominators by which the total cycle counts are scaled.

In theory, AVX-512 should perform twice as fast as AVX2 in terms of scaled cycle counts, because AVX-512 processes twice as many elements in one batch. Performing linear regression on the data sets in Figure 2 with `gnuplot` indicated that AVX-512 is faster than AVX2 by a factor of approximately 2.17 in practice.

We disabled Simultaneous multithreading (SMT) for all experiments in this paper. While we used multiple cores during benchmarking to find the best splitting strategy, the results in Table 1 (and later in Table 2) were ran on a single core.

3.3 Related Work

The cycle counts in this subsection are reported for $m = 251$ unless otherwise noted, since it is a common field size in the literature to approach the 128-

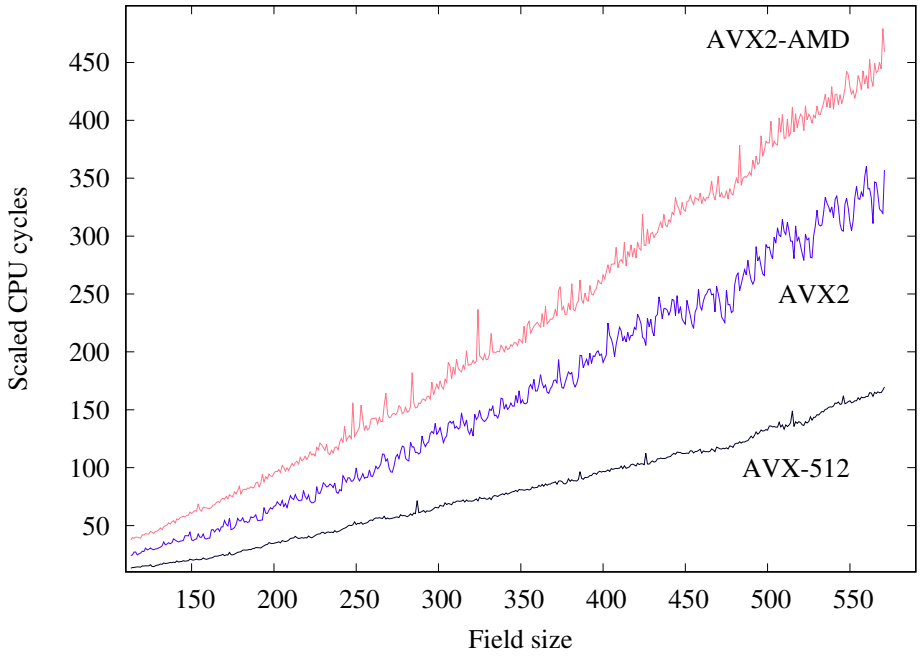

Fig. 2. Binary field multiplication performance in scaled CPU cycles.

Table 1. Selective binary field multiplication performance in scaled CPU cycles.

m	113	131	163	191	193	233	239	251	283	359	409	431	571
AVX-512	13	16	22	30	31	44	44	51	59	82	99	104	170
AVX2	24	30	43	54	55	84	77	91	121	166	216	214	351
AVX2-AMD	37	47	67	83	89	114	116	132	153	224	274	300	459
NEON	228	290	425	523	534	708	724	805	928	1340	1659	1819	2953

bit security level and therefore provides the best basis for comparison. We first note that the comparison here (and later in Section 4.5) to previous work is not without caveats, due to different computation models (i.e. parallel vs. serial) and ISA availability over time.

Aranha et al. [4] benchmarked several field operations on three different Intel platforms (Core 2 65nm, Core 2 45nm, and Core i7 45nm) Their best multiplication result was 323 cycles, achieved on the Core i7 platform with López-Dahab multiplication.

Taverne et al. [31, 32] benchmarked López-Dahab multiplication on an Intel Westmere Core i5 32nm processor, reporting 338 cycles for code compiled with GCC and 429 cycles for code compiled with ICC (the Intel C++ Compiler), and achieved a speedup to 161 cycles (GCC) and 159 cycles (ICC) by performing Karatsuba multiplication with the new carry-less multiplication instruction.

Câmara et al. [9] report timings for an ARM Cortex-A8, Cortex-A9, and Cortex-A15 processor, with code compiled with GCC. López-Dahab multiplication takes 671 cycles on A8, 774 on A9, and 412 on A15; their contribution is the Karatsuba/NEON/`VMULL` multiplication algorithm, which takes 385 cycles on A8, 491 on A9, and 317 on A15.

Oliveira et al. [25] benchmarked Karatsuba multiplication with carry-less multiplication on Intel Sandy Bridge for field size $m = 254$ and achieved 94 cycles with GCC (and report similar results for ICC).

Seo et al. [27] achieve 57 cycles for $m = 251$ and 153 cycles for $m = 571$ on ARMv8 by using the 64-bit polynomial multiplication instruction `PMULL` instead of the 8-bit polynomial multiplication instruction `VMULL` used in [9].

4 Elliptic Curve Arithmetic

A non-supersingular elliptic curve E over the binary finite field $\mathbb{F}_{2^m}$ is a set of points $(x, y) \in \mathbb{F}_{2^m}$ satisfying the short Weierstrass equation:

$$E(\mathbb{F}_{2^m}) : y^2 + xy = x^3 + ax^2 + b$$

where the parameters $a, b \in \mathbb{F}_{2^m}$ and the set of points in $E(\mathbb{F}_{2^m})$ is an additive Abelian group with identity element as the point at infinity $\mathcal{O}$ i.e. $P + \mathcal{O} = \mathcal{O} + P = P$ and the inverse element at $-P = (x_1, x_1 + y_1)$. For the sake of simplicity, the remainder of this paper will refer to this curve form as *short curves*.

4.1 Scalar Multiplication

The elliptic curve point multiplication is seen as the most critical and compute-intensive task, therefore a lot of emphasis is given on improving the algorithms for efficiency and security. For a given ℓ-bit scalar $k \in \mathbb{Z}$ and point $P \in E$, the point multiplication can be formulated as:

$$kP = \sum_{i=0}^{\ell-1} k_i 2^i P$$

where k_i is the i_{th} bit of the scalar. This naive method, a.k.a. binary method, scans the bits of the scalar one at a time, performing double operations for each $\ell - 1$ bits in addition to the add operation where $k_i \neq 0$.

An alternate approach was proposed by Montgomery [24]: where the same number of double-and-add operations are repeated in each step. Although—in terms of computational time—it seems more expensive as compared to the naive method. However, the advantage is the resistance against side-channel analysis by removing the conditional branches depending on the weight of scalar bits.

For performing fast ladder multiplication, differential addition and doubling is applied, which computes $P_1 + P_2$ from P_1, P_2 and $P_2 - P_1$ and similarly

$2P_1$ from P_1 for the doubling. The relation $P_2 - P_1 = P$ here is the invariant, i.e. the difference of these two points is known and constant throughout the ladder step. Short curves have a nice property, by applying suitable coordinate transformation, the knowledge of only x-coordinate is sufficient to perform the entire ladder step. This considerably improves the performance, since there is no need to evaluate the intermediate results of the y-coordinate at each ladder step.

An important consideration—for the performance optimization of EC double and add operations—is the representation of point coordinate system. The use of projective coordinates in most cases is preferred over the affine coordinates, due to the heavy inversion operations involved in the later. A number of coordinate systems, in the context of binary curves, have been studied such as lambda [25], Jacobian [12], and homogeneous projective [1]. Among them, López and Dahab [21] is a popular choice in binary elliptic curves, with various performance improvements proposed [2, 20, 7]. For a projective form of the short curves equation defined as:

$$E(\mathbb{F}_{2^m}) : Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4$$

the LD-projective point $(X_1 : Y_1 : Z_1)$ is equivalent to the affine point $(X_1/Z_1 : Y_1/Z_1^2)$, where the inverse is $(X_1 : X_1Z_1 + Y_1 : Z_1)$. It is further assumed that $Z_1 \neq 0$ and $\mathcal{O}$ is not among the points.

4.2 Differential Montgomery Ladder

For short curves, Bernstein and Lange[7] provide a very efficient differential addition and doubling (*ladder step*) formulas, originally derived from [29]. The representation, known as XZ, is a variant of the original LD coordinates. For each scalar bit k_i—assuming the invariant $P_2 - P_1 = P$ is known—the fast differential addition and doubling takes 5 multiplications, 4 squares and 1 multiplication by the constant $\sqrt{b}$. As mentioned by [14], this is the best case bound achieved for Montgomery ladder also on other forms of curves, such as Hessian [15], Huff [13], and Edwards [7], which means that choice of curve form is mostly irrelevant in terms of the computational cost. It is still possible to achieve slight performance gains over this bound by leveraging efficient forms of subfield curve constants [3].

Given $P_1 \neq \pm P_2$, to compute the differential point addition $P_3 = P_1 + P_2 = (X_3 : Y_3 : Z_3)$ where $P_1 = (X_1 : Y_1 : Z_1)$, $P_2 = (X_2 : Y_2 : Z_2)$ and, point doubling $P_4 = 2P_2 = (X_4 : Y_4 : Z_4)$, the formulas are defined as follows.

$$\begin{aligned} A &= X_1Z_2, B = X_2Z_1, C = X_1^2, D = Z_1^2 \\ Z_3 &= (A + B)^2 \\ X_3 &= XZ_3 + AB \\ X_4 &= (C + \sqrt{b}D) \\ Z_4 &= CD \end{aligned}$$

[7] https://hyperelliptic.org/EFD/g12o/auto-shortw-xz.html

As previously mentioned, the Montgomery ladder differential addition and double method does not take into account the y-coordinate during the computation, however to get back to the affine coordinates we need to recover the y-coordinate. López and Dahab [22] presented a formula to retrieve the y-coordinate, which enabled the complete point multiplication using the Montgomery ladder method, i.e. also compute the resulting affine point. In our case we applied a further optimization to the original formula by setting $Z = 1$ since the invariant $P_2 - P_1 = P = (X : Y : Z)$ is fixed, the resulting simplified version is formulated as:

$$x_1 = \frac{xZ_2X_1}{xZ_2Z_1}$$

$$y_1 = y + \frac{(X_2 + xZ_2)((X_1 + xZ_1)(X_2 + xZ_2) + Z_1Z_2x^2 + Z_1Z_2y)}{Z_1Z_2x}$$

The cost of recovering y is 10 multiplications and 1 inversion, which is small, considering that it is performed just once at the end of the ladder.

4.3 Linear Maps

As a further optimization, we also considered replacing the multiplication by the curve constants, and x-coordinate of the generator point during the ladder step by a linear map. For a finite field $\mathbb{F}_{2^m}$ represented as m-dimensional vector space $(\mathbb{F}_2)^m$, the linear map $T : \mathbb{F}_{2^m} \mapsto \mathbb{F}_{2^m}$ for multiplication by a constant b is an $(m \times m)$ matrix B with entries in $\mathbb{F}_2$. The elements b_i in B can be pre-computed: for a basis α of degree m, $b_i = \alpha^i b$ for $0 \leq i < m$, where b is the element of vector space $(\mathbb{F}_2)^m$. The multiplication by an element $a \in \mathbb{F}_{2^m}$ can then be replaced by $b \cdot a = Ba$, where each bit in the product can be written as $\sum_{i:b_{i,j}=1} a_i$.

This essentially means that if there are l non-zero bits for each row of the matrix B, in total we need $(l-1)m$ XOR gates for the entire matrix. This is not the optimal number of gates, since common sub-expression elimination is not accounted for. For this purpose we tried optimization as suggested by Bernstein [6] which produces approximately $ml/(\log l - \log\log(l))$ XORs, in addition to m number of copies of the intermediate results.

For hardware implementations, this straight-line optimization makes sense. However, in the software case this also increases both register pressure and, more importantly, binary size. For the above ladder step formula, we applied the tooling from [6] to replace multiplications by both the curve constant and fixed generator x-coordinate by the straight-line code. The resulting binary far exceeded the L1 cache size, and benchmark results showed it takes away any performance gains when compared to generic finite field multiplication in the ladder step.

4.4 Implementation

Our bitsliced ECC layer builds on our finite field layer from Section 3. As a library, it exposes a single function `EC2_batch_keygen` that takes four arguments:

(1) `EC2_CURVE` pointer to an opaque object provided by the library to represent a specific curve; (2) `unsigned char` pointer `k`, where it is assumed the caller has filled this input with enough randomness for w scalars, and from which the caller will retrieve the scalars as an output; (3) `unsigned char` pointer `x` storing the resulting x-coordinate of the scalar multiplication by G; and similarly (4) `unsigned char` pointer `y` for the y-coordinate. Geared to ease real-world deployment discussed further in Section 5, since our application is *only* to fixed (generator) point scalar multiplication, `EC2_batch_keygen` internally makes a number of optimizations.

We assume the scalars are provided in bitsliced form, simply viewing the first w bits at `k` as bit 0 of the scalars, and so on for the rest of the bits. This imposes no practical requirement on the caller—they are just random bytes, only viewed differently.

As previously discussed, for the range of standardized curves under consideration in this work, the linear maps for constant multiplications are not efficient when bitslicing, hence we implement them as generic finite field multiplications with fixed pre-bitsliced form stored in the binary. The *exception* to this case is `curve2251`, which has small and sparse curve coefficients. Here we implemented a dedicated ladder step using a slightly different formula involving the curve coefficient b and replace the multiplication with a straight-line linear map. Since this curve is not fixed by a standard, we found the lexicographically smallest x such that the resulting generator satisfies $\text{ord}(G) = n$, the large prime subgroup order. We then replaced this multiplication (by x-coordinate `e`) in the ladder step with a straight-line linear map. Of course we could do the same for standardized curves, but this would violate interoperability.

After scalar multiplication, `EC2_batch_keygen` recovers the y-coordinate, since key generation is our motivating use case and not simply key agreement where the x-coordinate is sufficient. Our finite field layer uses Itoh-Tsujii [17] in the inversion step, where the addition chain for exponentiation is efficient and fixed based on the finite field degree. Our finite field layer also supplies efficient finite field squaring with a straight-line linear map.

Finally, `EC2_batch_keygen` converts the outputs for `k`, `x`, and `y` from bitsliced form to canonical form for consumption by linking applications. Note for our use case of key generation, there are no exceptions in the ladder step, the y recovery, or scalar corner cases. We completely control all scalars and points.

4.5 Benchmarking

Our tooling for the ECC layer generates and exposes an `EC2_CURVE` object for each fixed curve. We restrict to short curves with (1) no efficient endomorphism and (2) field degrees that are prime. Table 2 reports the performance across our four target architectures. Each curve was benchmarked using the best splitting strategy for its respective underlying field, as measured by the benchmarking process described in Section 3. Similar to Section 3, the reported cycle counts are scaled, dividing by w.

Table 2. ECC performance on four architectures in scaled CPU cycles.

Curve	AVX-512	AVX2	AVX2-AMD	NEON
`sect113r1`	9547	18074	27470	153944
`sect113r2`	9540	17962	27487	153948
`sect131r1`	13684	26821	40478	227765
`sect131r2`	13639	26856	40466	228168
`sect163r1`	22849	45231	70046	427274
`sect163r2`	23175	46826	70413	448744
`c2pnb163v1`	23005	45017	74888	435651
`c2pnb163v2`	22881	45490	74413	426473
`c2pnb163v3`	22805	45380	74422	429631
`c2tnb191v1`	36094	66799	102005	632907
`c2tnb191v2`	36262	64963	101235	617958
`c2tnb191v3`	35680	64454	101325	629464
`sect193r1`	37899	67325	109376	639270
`sect193r2`	37936	67730	109434	640406
`sect233r1`	65491	125804	167761	1013458
`c2tnb239v1`	67144	119490	175914	1079930
`c2tnb239v2`	67105	117703	174388	1085560
`c2tnb239v3`	67181	120304	175676	1063116
`curve2251`	57756	106391	146031	870376
`sect283r1`	105304	218130	272544	1595423
`c2tnb359v1`	186680	362665	504219	2961857
`sect409r1`	260619	546690	697021	4229741
`c2tnb431r1`	283319	567608	780886	4812995
`sect571r1`	627668	1303759	1629335	10676160

For the sake of discussion, our results show it is more efficient to use canonical representations of the curve constants, including curve coefficients and x-coordinate of the generator point. This saves in terms of computational cost during the differential addition step, allowing `curve2251` to significantly outperform curves at comparable field sizes.

Finally, we briefly compare with selective results from the literature. Bernstein [5] reports 314K (scaled, SSE2, $w = 128$) Core 2 cycles for the binary Edwards curve `BBE251`. Aranha et al. [4] report 537K, 793K, and 4.4M Core i7 cycles for curves `curve2251`, `B-283`, and `B-571`. Taverne et al. [32] report 225K Sandy Bridge cycles for `curve2251` in constant time, 100K cycles for `B-233` in non constant time, and 349K cycles for `B-409` in non constant time. Oliveira et al. [25] report 114K Sandy Bridge cycles for a 254-bit curve in constant time, yet with a non-standard composite extension. Câmara et al. [9] report 511K, 866K, and 4.2M ARM Cortex-A15 cycles for constant time `curve2251`, `B-283`, and `B-571`. Oliveira et al. [26] report 46K Skylake cycles for a 254-bit curve in constant time, yet again with a non-standard composite extension.

Generally, even without focusing on a single curve yet imposing the batch computation requirement, the results in Table 2 compare very favorably with the existing literature. Considering it is automated tooling that generates the finite

field layer, and the ECC layer utilizes a stock ladder with no fast endomorphisms or precomputation, this demonstrates the tooling has wide applicability and can provide a strong baseline for performance comparison.

5 `ENGINE` implementation

As mentioned in Section 2, we integrate our bitsliced implementation in OpenSSL through an `ENGINE`, dubbed `libbecc`, modeled after `libsuola` [33].

We defer to [33] for a detailed description of the `ENGINE` framework and how it integrates with the OpenSSL architecture, while in this section we provide an overview of the design of `libbecc` in comparison to `libsuola`.

Our `ENGINE` provides implementations for most of the named[8] binary curves defined in OpenSSL and adds dedicated support to `curve2251`; this is achieved building on top of the work described in Sections 3 and 4, which provide the actual batch implementation for elliptic curve and binary field operations. The `libbecc` code mainly provides an interface to query the underlying layers and to dispatch to the relevant codepath when, through the OpenSSL library, an application requests a cryptographic operation that requires computation over a supported curve and field.

5.1 Providers

The other fundamental function of `libbecc` is to implement the logic to maintain a state for each performed batch operation, so that following requests can be served from the precomputed results rather than issuing a new batch operation.

We achieve support for batch ECC operations using the `ENGINE` API to register `libbecc` as the default `EC_KEY_METHOD`: by doing so our `ENGINE` is activated for any operation involving an `EC_KEY` object, including ECDH and ECDSA key generation, ECDH shared secret derivation and ECDSA signature generation and verification. `libbecc` retains a reference to the default OpenSSL `EC_KEY_METHOD`, which is used to bypass our `ENGINE` for unsupported curves or for operations such as ECDH derivation or ECDSA verification: these operations are not supported by our bitsliced code, limited to "fixed point" scalar multiplications, i.e., limited to scalar multiplications by the conventional generator point for a given curve.

Following `libsuola` approach and terminology, `libbecc` supports the notion of multiple providers to interface with the OpenSSL API, by providing a minimum set of functions to:

[8] The term "named" here is used in contrast with curves described by arbitrary parameters: usage in real-world applications is dominated by curves that have been assigned code points as part of standards, delivering both security assurances on the cryptographic features and security evaluation of the group defined by the specified set of parameters, and saving the users from the need of performing expensive validation of the group parameters during curve negotiation.

- match an `EC` object with any of the supported curves, returning either *unsupported* or a provider specific integer identifier for the specific curve;
- generate one key for a given curve identifier, returning a secret random scalar and the corresponding public point;
- perform the setup step of ECDSA signature generation for a curve identifier, returning the modular inverse of a secret nonce scalar and the corresponding `r` component of a `(r,s)` ECDSA signature.

Providers can then internally differ on the way the batch logic is implemented to support a bitsliced scalar multiplication.

Provider: `serial`. Specifically, we instantiate one provider, dubbed `serial`, that stores the internal state and buffers for key generation and signature setup with a thread local model. In this model, each thread of an OpenSSL application loading our `ENGINE` stores its own local state and buffers, and the batch results are not shared across separate threads.

In our design, we opted to perform all the operations required by a specific cryptosystem operation during the batch computation, including the conversion of raw binary buffers in OpenSSL `BIGNUM` objects, and, during the batch computation for `sign_setup`, also the batch inversion of the original nonces. To improve performance, we implemented the batch inversion using the so-called Montgomery trick [24, Sect. 10.3.1] which allows to compute simultaneously the inverses of n elements at the cost of 1 inversion and $(3n-1)$ multiplications [28, Sect. 3.1].

The design of the `serial` provider is the most straightforward strategy to implement batch operations in OpenSSL. It avoids synchronization issues and allows us to evaluate the performance of our implementation compared to the baseline upstream implementation with classic benchmarking tools, as they usually consist of a serial loop of repeated operations.

On the other hand, we acknowledge that this is not the optimal implementation for real-world high-load multi-threaded or multi-process applications, which would actually benefit the most from bitsliced operations, saving memory resources and minimizing the number of batch operations to run, if a more clever logic to share the results of batch operations were implemented. In particular, although of limited academic value for this paper, it would be interesting to add a provider supporting inter-process communication, to have a separate system-wide *singleton* service in charge of running the batch operations and storing the results, while applications using `libbecc` would simply request a fresh result from the service. Such design would minimize memory consumption and the number of batch operations across the whole system, and it would also provide a stronger security model to protect the state and buffers in memory, as they would be stored in a separate process space. Leveraging the access control capabilities of the underlying operating system, this would be inaccessible from a compromised application.

5.2 Benchmarking

To evaluate the actual practical impact of the presented improvements, we instantiated a benchmarking application built on top of OpenSSL 1.1.1. Table 3 reports the average number of CPU clock cycles to compute a single ECDH/ECDSA key generation and ECDSA signature generation. We compare the results recorded against the default implementation with a run of the same application after loading `libbecc` at runtime; due to space constraints we limit this analysis to the AVX2 platform. For comparison, we also include measurements relative to operations on top of popular prime curves, namely ECDH/ECDSA over `secp256r1` (a.k.a. NIST `P-256`), and EDDSA over `ED25519` and `ED448`.

The benchmarking application consecutively runs 2^{16} operations for each curve, recording the number of elapsed CPU cycles after each operation; Table 3 reports the average of such measurements. It should be noted that, for the default implementation, each measure for a given operation on a given curve is relatively close to the average reported on the table; for the `libbecc` implementation instead, due to the nature of batch operations, we record execution time spikes when a new batch is computed followed by a relatively low (between nine and sixteen thousands of CPU cycles, depending on the field size of the curve) plateau for each following operation until the batch is consumed.

6 Conclusion

In this paper, we developed a tool to optimize bitsliced binary field arithmetic that can potentially support any architecture. Guided by benchmarking statistics, at the finite field layer the tool tries different polynomial splitting strategies to arrive at the performance-optimal configuration on a given platform. Building on this layer, we developed a second tool that implements an ECC layer for a given binary curve, and performs very competitively; e.g. 58K AVX-512 (scaled) cycles for constant-time fixed point scalar multiplication on `curve2251`. Building on both these results, our last layer links OpenSSL with the output, overcoming the restriction of batch computation at the application level. Our approach in `libbecc` seamlessly couples applications with the batch computation results, facilitating real-world deployment of bitsliced public key cryptography software.

Future work. Our ECC layer is specific to short curves; a natural direction is to extend with support for other binary curve forms, even using the birational equivalence with short curves where applicable to maintain compatibility with existing (legacy, X9.62) standards exposed by OpenSSL through `libbecc`. Lastly, the architecture of `libbecc` has several applications outside binary ECC. Exploring similar functionality for traditional SIMD instead of bitslicing is another research direction, providing batch computation for curves over prime fields. Such an implementation would have much lower batch sizes, but potentially far greater performance since it relaxes register and memory pressure.

Acknowledgments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and

Table 3. OpenSSL performance on AVX2: average CPU cycles for operations on selected curves.

Curve	Average CPU cycles per operation			
	Key generation		Signature generation	
	default	`libbecc`	default	`libbecc`
`sect113r1`	309998	26758 (11.6x)	323163	25049 (12.9x)
`sect113r2`	309892	26810 (11.6x)	323586	25043 (12.9x)
`sect131r1`	515314	37617 (13.7x)	533484	35950 (14.8x)
`sect131r2`	519635	37293 (13.9x)	540221	35711 (15.1x)
`c2pnb163v1`	699094	54717 (12.8x)	718671	52785 (13.6x)
`c2pnb163v2`	690930	54603 (12.7x)	710865	52653 (13.5x)
`c2pnb163v3`	700345	54432 (12.9x)	722117	52709 (13.7x)
`sect163r1`	690258	53996 (12.8x)	719847	52548 (13.7x)
`sect163r2`	697992	54875 (12.7x)	725706	52635 (13.8x)
`c2tnb191v1`	673839	74407 (9.1x)	694368	72989 (9.5x)
`c2tnb191v2`	668479	74308 (9.0x)	695895	72811 (9.6x)
`c2tnb191v3`	669240	73836 (9.1x)	697687	73037 (9.6x)
`sect193r1`	762628	77378 (9.9x)	803603	76853 (10.5x)
`sect193r2`	758937	77109 (9.8x)	800200	76908 (10.4x)
`sect233r1`	940852	133436 (7.1x)	985741	133941 (7.4x)
`c2tnb239v1`	966659	127149 (7.6x)	1008116	126843 (7.9x)
`c2tnb239v2`	960048	126222 (7.6x)	1004913	126053 (8.0x)
`c2tnb239v3`	961976	125478 (7.7x)	1008270	125681 (8.0x)
`curve2251`	1139439	118368 (9.6x)	1198634	118661 (10.1x)
`ED25519`	130295	130254 (1.0x)	131174	129597 (1.0x)
`secp256r1`	35805	36741 (1.0x)	69228	68921 (1.0x)
`sect283r1`	1631437	226075 (7.2x)	1700907	225973 (7.5x)
`c2tnb359v1`	2016295	377226 (5.3x)	2126677	374781 (5.7x)
`sect409r1`	2731705	552476 (4.9x)	2880190	551485 (5.2x)
`c2tnb431r1`	2968140	587026 (5.1x)	3125245	579913 (5.4x)
`ED448`	960595	957772 (1.0x)	969825	969300 (1.0x)
`sect571r1`	6283098	1359731 (4.6x)	6624862	1311432 (5.1x)

innovation programme (grant agreement No 804476). The second author was supported in part by the Tuula and Yrjö Neuvo Fund through the Industrial Research Fund at Tampere University of Technology.

References

1. Agnew, G.B., Mullin, R.C., Vanstone, S.A.: An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. IEEE Journal on Selected Areas in Communications 11(5), 804–813 (1993), https://doi.org/10.1109/49.223883
2. Al-Daoud, E., Mahmod, R., Rushdan, M., Kiliçman, A.: A new addition formula for elliptic curves over GF(2^n). IEEE Trans. Computers 51(8), 972–975 (2002), https://doi.org/10.1109/TC.2002.1024743
3. Aranha, D.F., Azarderakhsh, R., Karabina, K.: Efficient software implementation of laddering algorithms over binary elliptic curves. In: Ali, S.S., Danger, J., Eisen-

barth, T. (eds.) Security, Privacy, and Applied Cryptography Engineering - 7th International Conference, SPACE 2017, Goa, India, December 13-17, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10662, pp. 74–92. Springer (2017), https://doi.org/10.1007/978-3-319-71501-8_5

4. Aranha, D.F., López, J., Hankerson, D.: Efficient software implementation of binary field arithmetic using vector instruction sets. In: Abdalla, M., Barreto, P.S.L.M. (eds.) Progress in Cryptology - LATINCRYPT 2010, First International Conference on Cryptology and Information Security in Latin America, Puebla, Mexico, August 8-11, 2010, Proceedings. Lecture Notes in Computer Science, vol. 6212, pp. 144–161. Springer (2010), https://doi.org/10.1007/978-3-642-14712-8_9
5. Bernstein, D.J.: Batch binary Edwards. In: Halevi, S. (ed.) Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5677, pp. 317–336. Springer (2009), https://doi.org/10.1007/978-3-642-03356-8_19
6. Bernstein, D.J.: Optimizing linear maps modulo 2. In: SPEED-CC: Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers, Workshop Record. pp. 3–18 (2009), http://cr.yp.to/papers.html#linearmod2
7. Bernstein, D.J., Lange, T., Farashahi, R.R.: Binary Edwards curves. In: Oswald, E., Rohatgi, P. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5154, pp. 244–265. Springer (2008), https://doi.org/10.1007/978-3-540-85053-3_16
8. Brumley, B.B., Page, D.: Bit-sliced binary normal basis multiplication. In: Antelo, E., Hough, D., Ienne, P. (eds.) 20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011. pp. 205–212. IEEE Computer Society (2011), https://doi.org/10.1109/ARITH.2011.36
9. Câmara, D.F., Gouvêa, C.P.L., López, J., Dahab, R.: Fast software polynomial multiplication on ARM processors using the NEON engine. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E.R., Xu, L. (eds.) Security Engineering and Intelligence Informatics - CD-ARES 2013 Workshops: MoCrySEn and SeCIHD, Regensburg, Germany, September 2-6, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8128, pp. 137–154. Springer (2013), https://doi.org/10.1007/978-3-642-40588-4_10
10. Cenk, M.: Karatsuba-like formulae and their associated techniques. J. Cryptographic Engineering 8(3), 259–269 (2018), https://doi.org/10.1007/s13389-017-0155-8
11. Cenk, M., Hasan, M.A.: Some new results on binary polynomial multiplication. J. Cryptographic Engineering 5(4), 289–303 (2015), https://doi.org/10.1007/s13389-015-0101-6
12. Chudnovsky, D., Chudnovsky, G.: Sequences of numbers generated by addition in formal groups and new primality and factorization tests. Advances in Applied Mathematics 7(4), 385–434 (1986), http://dx.doi.org/10.1016/0196-8858(86)90023-0
13. Devigne, J., Joye, M.: Binary Huff curves. In: Kiayias, A. (ed.) Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6558, pp. 340–355. Springer (2011), https://doi.org/10.1007/978-3-642-19074-2_22

14. Farashahi, R.R., Hosseini, S.G.: Differential addition on binary elliptic curves. In: Duquesne, S., Petkova-Nikova, S. (eds.) Arithmetic of Finite Fields - 6th International Workshop, WAIFI 2016, Ghent, Belgium, July 13-15, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10064, pp. 21–35 (2016), https://doi.org/10.1007/978-3-319-55227-9_2
15. Farashahi, R.R., Joye, M.: Efficient arithmetic on Hessian curves. In: Nguyen, P.Q., Pointcheval, D. (eds.) Public Key Cryptography - PKC 2010, 13th International Conference on Practice and Theory in Public Key Cryptography, Paris, France, May 26-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6056, pp. 243–260. Springer (2010), https://doi.org/10.1007/978-3-642-13013-7_15
16. Find, M.G., Peralta, R.: Better circuits for binary polynomial multiplication. IEEE Trans. Computers 68(4), 624–630 (2019), https://doi.org/10.1109/TC.2018.2874662
17. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. Inform. and Comput. 78(3), 171–177 (1988), https://doi.org/10.1016/0890-5401(88)90024-7
18. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings. Lecture Notes in Computer Science, vol. 5747, pp. 1–17. Springer (2009), https://doi.org/10.1007/978-3-642-04138-9_1
19. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation 48(177), 203–209 (1987)
20. Lange, T.: A note on López-Dahab coordinates. IACR Cryptology ePrint Archive 2004, 323 (2004), http://eprint.iacr.org/2004/323
21. López, J., Dahab, R.: Improved algorithms for elliptic curve arithmetic in GF(2^n). In: Tavares, S.E., Meijer, H. (eds.) Selected Areas in Cryptography '98, SAC'98, Kingston, Ontario, Canada, August 17-18, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1556, pp. 201–212. Springer (1998), https://doi.org/10.1007/3-540-48892-8_16
22. López, J., Dahab, R.: Fast multiplication on elliptic curves over GF(2^m) without precomputation. In: Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1717, pp. 316–327. Springer (1999), https://doi.org/10.1007/3-540-48059-5_27
23. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings. Lecture Notes in Computer Science, vol. 218, pp. 417–426. Springer (1985), https://doi.org/10.1007/3-540-39799-X_31
24. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comp. 48(177), 243–264 (1987), https://doi.org/10.2307/2007888
25. Oliveira, T., López, J., Aranha, D.F., Rodríguez-Henríquez, F.: Two is the fastest prime: lambda coordinates for binary elliptic curves. J. Cryptographic Engineering 4(1), 3–17 (2014), https://doi.org/10.1007/s13389-013-0069-z
26. Oliveira, T., López, J., Rodríguez-Henríquez, F.: The Montgomery ladder on binary elliptic curves. J. Cryptographic Engineering 8(3), 241–258 (2018), https://doi.org/10.1007/s13389-017-0163-8
27. Seo, H., Liu, Z., Nogami, Y., Choi, J., Kim, H.: Binary field multiplication on ARMv8. Security and Communication Networks 9(13), 2051–2058 (2016), https://doi.org/10.1002/sec.1462

28. Shacham, H., Boneh, D.: Improving SSL handshake performance via batching. In: Naccache, D. (ed.) Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2020, pp. 28–43. Springer (2001), https://doi.org/10.1007/3-540-45353-9_3
29. Stam, M.: On Montgomery-Like representations for elliptic curves over GF(2^k). In: Desmedt, Y. (ed.) Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2567, pp. 240–253. Springer (2003), https://doi.org/10.1007/3-540-36288-6_18
30. Stebila, D., Mosca, M.: Post-quantum key exchange for the Internet and the Open Quantum Safe project. In: Avanzi, R., Heys, H.M. (eds.) Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10532, pp. 14–37. Springer (2016), https://doi.org/10.1007/978-3-319-69453-5_2
31. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Software implementation of binary elliptic curves: Impact of the carry-less multiplier on scalar multiplication. In: Preneel, B., Takagi, T. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6917, pp. 108–123. Springer (2011), https://doi.org/10.1007/978-3-642-23951-9_8
32. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. J. Cryptographic Engineering 1(3), 187–199 (2011), https://doi.org/10.1007/s13389-011-0017-8
33. Tuveri, N., Brumley, B.B.: Start your ENGINEs: dynamically loadable contemporary crypto. In: IEEE Secure Development Conference, SecDev 2019, McLean, VA, USA, September 25-27, 2019. IEEE Computer Society (2019), https://eprint.iacr.org/2018/354
34. Wiggers, T.: Energy-efficient ARM64 cluster with cryptanalytic applications: 80 cores that do not cost you an ARM and a leg. In: Progress in Cryptology - LATINCRYPT 2017 - 5th International Conference on Cryptology and Information Security in Latin America, La Habana, Cuba, September 20-22, 2017, Proceedings. LNCS, Springer (2017), https://eprint.iacr.org/2018/888