Faster software for fast endomorphisms

COSADE 2015, Berlin

B. B. Brumley

Department of Pervasive Computing
Tampere University of Technology, Finland
billy.brumley AT tut.fi

13 Apr 2015

TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

GLV curves

CRYPTO 2001

Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms

Robert P. Gallant[1], Robert J. Lambert[1], and Scott A. Vanstone[1,2]

[1] Certicom Research, Canada
{rgallant,rlambert,svanstone}@certicom.com
[2] University of Waterloo, Canada

Curve endomorphisms

Example 1. Let E be an elliptic curve defined over $\mathbb{F}_q$. For each $m \in \mathbb{Z}$ the *multiplication by m* map $[m] : E \to E$ defined by $P \mapsto mP$ is an endomorphism defined over $\mathbb{F}_q$. A special case is the *negation* map defined by $P \mapsto -P$.

. . .

Example 4 (§7.2.3 of [5]). Let $p \equiv 1 \pmod 3$ be a prime, and consider the elliptic curve

$$E_2 \; : \; y^2 = x^3 + b \tag{2}$$

defined over $\mathbb{F}_p$. Let $\beta \in \mathbb{F}_p$ be an element of order 3. Then the map $\phi : E_2 \to E_2$ defined by $(x, y) \mapsto (\beta x, y)$ and $\mathcal{O} \mapsto \mathcal{O}$ is an endomorphism defined over $\mathbb{F}_p$. If $P \in E(\mathbb{F}_p)$ is a point of prime order n, then ϕ acts on $\langle P \rangle$ as a multiplication map $[\lambda]$, where λ is an integer satisfying $\lambda^2 + \lambda \equiv -1 \pmod n$. Note that $\phi(Q)$ can be computed using only one multiplication in $\mathbb{F}_p$.

Scalar decomposition

The problem we consider is that of computing kP for k selected uniformly at random from the interval $[1, n-1]$. The basic idea of the paper is as follows. Suppose that we can efficiently write $k = k_1 + k_2\lambda \bmod n$, where $k_1, k_2 \in [0, \lceil\sqrt{n}\,\rceil]$ (see §4). Then we have

$$\begin{aligned} kP &= (k_1 + k_2\lambda)P \\ &= k_1P + k_2(\lambda P) \\ &= k_1P + k_2\phi(P). \end{aligned} \tag{6}$$

Now (6) can be computed using any of the 'simultaneous multiple exponentiation' type algorithms[1], the simplest of which we review below. In the following,

Standard curves in OpenSSL

```
$ openssl ecparam -list_curves
  secp160k1 : SECG curve over a 160 bit prime field
  secp160r1 : SECG curve over a 160 bit prime field
  secp160r2 : SECG/WTLS curve over a 160 bit prime field
  secp192k1 : SECG curve over a 192 bit prime field
  secp224k1 : SECG curve over a 224 bit prime field
  secp224r1 : NIST/SECG curve over a 224 bit prime field
  secp256k1 : SECG curve over a 256 bit prime field
  prime192v1: NIST/X9.62/SECG curve over a 192 bit prime field
  prime192v2: X9.62 curve over a 192 bit prime field
  prime192v3: X9.62 curve over a 192 bit prime field
  prime239v1: X9.62 curve over a 239 bit prime field
  prime239v2: X9.62 curve over a 239 bit prime field
  prime239v3: X9.62 curve over a 239 bit prime field
  prime256v1: X9.62/SECG curve over a 256 bit prime field
  ...
```

Bitcoin

https://en.bitcoin.it/wiki/Secp256k1

Search

Navigation

- Main page
- FAQ
- Help
- Forums
- Chatrooms
- Community portal
- Recent changes

Page Discussion Read View source View history

Secp256k1

secp256k1 refers to the parameters of the ECDSA curve used in Bitcoin, and is defined in *Standards for Efficient Cryptography (SEC)* (Certicom Research, http://perso.univ-rennes1.fr/sylvain.duquesne/master/standards/sec2_final.pdf).

secp256k1 was almost never used before Bitcoin became popular, but it is now gaining in popularity due to its several nice properties. Most commonly-used curves have a random structure, but secp256k1 was constructed in a special non-random way which allows for especially efficient computation. As a result, it is often more than 30% faster than other curves if the implementation is sufficiently optimized. Also, unlike the popular NIST curves, secp256k1's constants were selected in a predictable way, which significantly reduces the possibility that the curve's creator inserted any sort of backdoor into the curve.

ECC in OpenSSL

```
struct ec_method_st {
int flags; /* Various method flags */
...
/* used by EC_POINT_add, EC_POINT_dbl, ECP_POINT_invert: */
int (*add)(const EC_GROUP *, EC_POINT *r, const EC_POINT *a, const EC_POINT *b, BN_CTX *);
int (*dbl)(const EC_GROUP *, EC_POINT *r, const EC_POINT *a, BN_CTX *);
int (*invert)(const EC_GROUP *, EC_POINT *, BN_CTX *);

/* used by EC_POINT_is_at_infinity, EC_POINT_is_on_curve, EC_POINT_cmp: */
int (*is_at_infinity)(const EC_GROUP *, const EC_POINT *);
int (*is_on_curve)(const EC_GROUP *, const EC_POINT *, BN_CTX *);
int (*point_cmp)(const EC_GROUP *, const EC_POINT *a, const EC_POINT *b, BN_CTX *);

/* used by EC_POINT_make_affine, EC_POINTs_make_affine: */
int (*make_affine)(const EC_GROUP *, EC_POINT *, BN_CTX *);
int (*points_make_affine)(const EC_GROUP *, size_t num, EC_POINT *[], BN_CTX *);

/* used by EC_POINTs_mul, EC_POINT_mul, EC_POINT_precompute_mult, EC_POINT_have_precompute_mult
 * (default implementations are used if the 'mul' pointer is 0): */
int (*mul)(const EC_GROUP *group, EC_POINT *r, const BIGNUM *scalar,
size_t num, const EC_POINT *points[], const BIGNUM *scalars[], BN_CTX *);
int (*precompute_mult)(EC_GROUP *group, BN_CTX *);
int (*have_precompute_mult)(const EC_GROUP *group);

/* 'field_mul', 'field_sqr', and 'field_div' can be used by 'add' and 'dbl' so that
 * the same implementations of point operations can be used with different
 * optimized implementations of expensive field operations: */
int (*field_mul)(const EC_GROUP *, BIGNUM *r, const BIGNUM *a, const BIGNUM *b, BN_CTX *);
int (*field_sqr)(const EC_GROUP *, BIGNUM *r, const BIGNUM *a, BN_CTX *);
int (*field_div)(const EC_GROUP *, BIGNUM *r, const BIGNUM *a, const BIGNUM *b, BN_CTX *);
...
} /* EC_METHOD */;
```

Scalar multiplication in OpenSSL

OpenSSL has a very general (and efficient) multi-scalar API:

```
/** Computes r = generator * n sum_{i=0}^{num-1} p[i] * m[i]
 *  \param  group  underlying EC_GROUP object
 *  \param  r      EC_POINT object for the result
 *  \param  n      BIGNUM with the multiplier for the group generator (optional)
 *  \param  num    number futher summands
 *  \param  p      array of size num of EC_POINT objects
 *  \param  m      array of size num of BIGNUM objects
 *  \param  ctx    BN_CTX object (optional)
 *  \return 1 on success and 0 if an error occured
 */
int EC_POINTs_mul(const EC_GROUP *group, EC_POINT *r, const BIGNUM *n,
                  size_t num, const EC_POINT *p[], const BIGNUM *m[], BN_CTX *ctx);
```

This paper

- Make a GLV method, and use our own `mul` function pointer
- The function decomposes scalars and wraps `EC_POINTs_mul`
- Deploy some side-channel countermeasures (2009, 2014 attacks)

Focus on minimally intrusive changes.

Regular scalar encodings

M. Joye and M. Tunstall (2009)

- NAF is sparse, but has "natural" zeroes. E.g. 31415:

$$(1, 0, 0, 0, 0, 0, -3, 0, 0, 3, 0, 0, -1, 0, 0, -1)$$

- We need to fix the weight.
- Need to use the same digit set to maintain compatibility.

Exponent Recoding and Regular Exponentiation Algorithms 341

Example 2. Again, with the example of exponent $n = 31415$, Algorithm 6 produces the equivalent representation $(1, 1, 1, 1, 1, \bar{1}, 1, \bar{1}, 1, \bar{1}, 1, 1, \bar{1}, 1, 1)_{\pm 2}$ for $m = 2$. For $m = 4$, it produces $(1, 3, 3, \bar{1}, \bar{1}, \bar{1}, 1, 3)_{\pm 4}$.

Table lookups

```
...
        int digit = wNAF[i][k];
        int is_neg;

        if (digit)
          {
          is_neg = digit < 0;

          if (is_neg)
            digit = -digit;

          if (is_neg != r_is_inverted)
            {
            if (!r_is_at_infinity)
              {
              if (!EC_POINT_invert(group, r, ctx)) goto err;
              }
            r_is_inverted = !r_is_inverted;
            }

          /* digit > 0 */

          if (r_is_at_infinity)
            {
            if (!EC_POINT_copy(r, val_sub[i][digit >> 1])) goto err;
            r_is_at_infinity = 0;
            }
          else
            {
            if (!EC_POINT_add(group, r, r, val_sub[i][digit >> 1], ctx)) goto err;
            }
          }
...
```

Software multiplexing

E. Käsper (2011)

Listing 1 A routine for choosing between two inputs `a` and `b` in constant time, depending on the selection bit `bit`.

```
int select (int a, int b, int bit) {
    /* -0 = 0, -1 = 0xff....ff */
    int mask = - bit;
    int ret = mask & (a^b);
    ret = ret ^ a;
    return ret;
}
```

Listing 2 A cache-timing resistant table lookup.

```
int do_lookup(int a[16], int bit[4]) {
    int t0[8], t1[4], t2[2];
    /* select values where the least significant bit of the index is bit[0] */
    t0[0] = select(a[0], a[1], bit[0]); t0[1] = select(a[2], a[3], bit[0]);
    t0[2] = select(a[4], a[5], bit[0]); t0[3] = select(a[6], a[7], bit[0]);
    t0[4] = select(a[8], a[9], bit[0]); t0[5] = select(a[10], a[11], bit[0]);
    t0[6] = select(a[12], a[13], bit[0]); t0[7] = select(a[14], a[15], bit[0]);
    /* select values where the second bit of the index is bit[1] */
    t1[0] = select(t[0], t[1], bit[1]); t1[1] = select(t[2], t[3], bit[1]);
    t1[2] = select(t[4], t[5], bit[1]); t1[3] = select(t[6], t[7], bit[1]);
    /* select values where the third bit of the index is bit[2] */
    t2[0] = select(t2[0], t[1], bit[2]); t2[1] = select(t2[2], t2[3], bit[2]);
    /* select the value where the most significant bit of the index is bit[3] */
    ret = select(t3[0], t3[1], bit[3]);
    return ret;
}
```

ECDH numbers

ops/sec, Haswell

curve	stock	GLV	GLV* +RNAF	GLV* +MUX	GLV* +RNAF+MUX
secp160r1	6824.1	—	6222.6	6171.9	6204.4 (-9.1%)
nistp192	5707.6	—	5317.4	5280.6	5198.8 (-8.9%)
nistp224	4077.2	—	3739.0	3785.5	3753.0 (-8.0%)
nistp256	3651.3	—	3296.1	3317.2	3319.5 (-9.1%)
secp160k1	6156.4	9292.0 (50.9%)	8173.8	8214.4	8175.9 (32.8%)
secp192k1	5181.2	7826.9 (51.1%)	6880.4	6864.7	6721.2 (29.7%)
secp224k1	3784.0	5527.7 (46.1%)	4955.4	5004.3	4891.6 (29.3%)
secp256k1	3265.8	4851.1 (48.5%)	4253.3	4276.7	4357.6 (33.4%)

ECDSA sign numbers

ops/sec, Haswell

curve	stock	RNAF	MUX	RNAF+MUX
secp160r1	19256.8	16247.6	16278.3	16244.6 (-15.6%)
nistp192	15968.0	13089.3	13099.0	13079.3 (-18.1%)
nistp224	12954.2	10259.1	10326.3	10348.5 (-20.1%)
nistp256	11067.5	8881.7	8896.7	8872.9 (-19.8%)
secp160k1	18970.1	16131.7	16118.3	16121.2 (-15.0%)
secp192k1	15629.7	12834.8	12819.6	12814.7 (-18.0%)
secp224k1	12676.9	10441.1	10383.2	10352.5 (-18.3%)
secp256k1	10760.9	8708.1	8687.7	8682.9 (-19.3%)

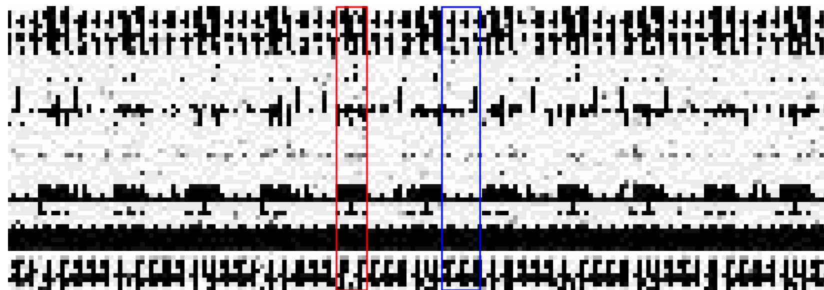
ECDSA verify numbers

ops/sec, Haswell

curve	stock	GLV
secp160r1	5526.9	—
nistp192	4623.8	—
nistp224	3353.6	—
nistp256	2912.5	—
secp160k1	5131.7	6676.5 (30.1%)
secp192k1	4251.0	5484.0 (29.0%)
secp224k1	3202.3	4175.2 (30.4%)
secp256k1	2730.8	3672.9 (34.5%)

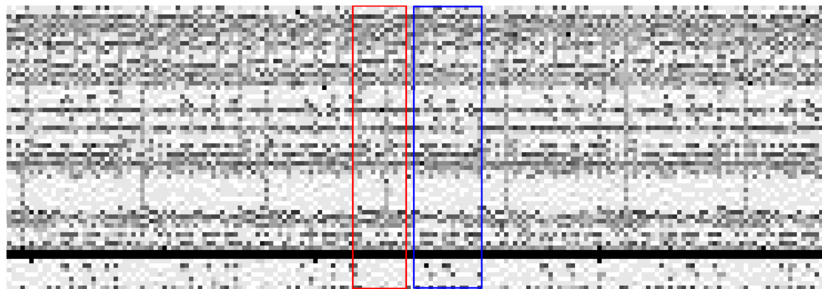
ECDH icache view (insecure)

ECDH icache view (secure)



ECDH dcache view (insecure)

ECDH dcache view (secure)



Conclusion

- secp256k1 *with* side-channel countermeasures now faster than nistp256 *without*
- SCA evaluation
- Still not fully constant time
- Three independent patches (two submitted so far)
- Thanks! Questions?

Open position

- Side-channel analysis project
- Seeking a postdoc or DSc student