A Survey of Tiny ECC - A Small Library for ECC in Configurable Wireless Networks, and its Optimizations

Ben Terner

Abstract—**Elliptic curves offer strong computational security for public key cryptography while requiring far smaller keys than RSA. In a world where interconnected devices trend toward miniaturization and security threats become more frequent, elliptic curves are especially appealing to implementors of embedded devices with strong security requirements and few computing resources.**

This survey will investigate the library TinyECC for implementing elliptic curve cryptography in embedded systems and devices with low computational power and the optimizations that it employs. TinyECC is a library designed for wireless sensor networks that allows its users to choose configure optimizations based on the computing resources available on a device and performance requirements.

I. Introduction

As the world fills with devices connected over wireless networks, the need arises for devices with strong cryptographic capabilities available at low power. Standard public key cryptography, which uses RSA or Diffie-Hellman, requires long keys and large computations that can become very memory intensive, possibly occupying a large portion of the computational abilities of the device. Elliptic curve cryptography achieves similar security to RSA while requiring *much shorter* keys and computation over *much smaller* integers, making it much better suited to devices with limited computational capabilities.

However, the problems on which ECC is built are still difficult and computationally intensive, with the potential to drain the power capabilities of the device, so implementing tools for ECC on small devices requires optimization to improve the efficiency of the computation. TinyECC aims to provide a low-power, portable, cross-platform, and computationally efficient implementation of cryptographic algorithms with optimizations suitable for low power devices. By understanding the performance of TinyECC and its performance, we can also gain insight into the bottlenecks in computing elliptic curve cryptography.

TinyECC implements the three standard elliptic curve cryptographic algorithms: Elliptic Curve Diffie-Hellman (ECDH) key agreement, Elliptic Curve Digital Signature Algorithm (ECDSA), and Elliptic Curve Integrated Encryption Scheme (ECIES).

II. Optimizations

To achieve its stated design goals, TinyECC [1] [2] employs a variety of optimizations to both improve performance and reduce code size. In general, however, the more complex algorithms trade efficiency for an increase in code size and RAM necessary, since they hold intermediate values in memory to speed up computation if they can be reused by similar operations rather than compute them as needed. Here, we detail the optimizations.

A. Barrett Reduction

Rather than perform modular reductions in the "textbook" way, using division, Barrett Reduction offers a method for computing a reduction using two multiplications and a small number of reductions via integers of the form 2^m. Although the textbook method is more efficient for a single modular reduction, Barrett Reduction provides performance benefits for many reductions modulo the same number by permitting pre-computation and reuse of intermediate values. Barrett Reduction is similar to Montgomery's modular multiplication method, although Montgomery multiplication requires converting the operands into Montgomery form and back, while Barrett Reduction trades off by requiring pre-computation.

TinyECC identifies the tradeoffs of Barrett Reduction in terms of speed in processing for size of code and occupation of RAM, since it needs to store intermediate values. We give Barrett's Algorithm in Algorithm 1 and describe it below.

Input: $p, b > 3, k = \lfloor \log_b p \rfloor + 1, 0 \leq z \leq b^{2k}, \mu = \lfloor b^{2k}/p \rfloor$
Output: $z \mod p$

$q \leftarrow \lfloor \lfloor z/b^{k-1} \rfloor \cdot \mu / b^{k+1} \rfloor$
$r \leftarrow (z \mod b^{k+1}) - (q \cdot p \mod b^{k+1})$
if $r < 0$ **then**
  $r \leftarrow r + b^{k+1}$
end if
while $r \geq p$ **do**
  $r \leftarrow r - p$
end while
return r

Algorithm 1: Barrett's Algorithm for modular reduction

The value b is a base suitably chosen for the modulus, and if it is set to some value $b = 2^L$ for an appropriate L, then division by b is just a right shift, making the operation highly efficient. The value $\mu = \lfloor b^{2k}/p \rfloor$ is dependent only on the modulus and not on the variable being reduced, so although its computation may be expensive (requiring the number of

times the prime modulus p divides b^{2k}), it can be reused for all modulus computations.

B. Hybrid Multiplication and Hybrid Squaring

The Hybrid Algorithm given by Gura et. al [3] is not reproduced here for its tediousness, but instead we give a description and discuss its advantages. The objective of Hybrid multiplication is to have an algorithm which is both fast and requires minimal storage; if the storage is small enough, efficient use of registers by the operating system and processor might improve speed of memory lookups; however, the main objective of a hybrid approach with TinyECC is to balance speed with memory requirements on devices with minimal storage.

The Hybrid algorithm attempts to find balance between the benefits of a row-wise multiplication and a column-wise multiplication. In a row-wise multiplication, one bit of the multiplier is kept constant as an AND is computed with each bit of the multiplicand, and partial products are summed in an accumulator for each bit of the multiplier. In a column-wise multiplication, columns of partial products are collected for the products $a_i \cdot b_j$, where $i + j = l$ for column l. At the end of each column, a k bit word is stored as part of the final product. Although this method requires very few registers, the access pattern to memory is not as regular (in row-wise multiplication, one needed only to increment a pointer to find the next bit or word of the multiplier and multiplicand) potentially slowing down the computation.

To achieve a balance between the row-wise and column-wise multiplication, the Hybrid algorithm combines both. At a high level, it uses column-wise multiplication as its "outer algorithm" and row-wise multiplication as its "inner algorithm" in order to compute columns comprising of the rows of partial products. The intuition is that the k-bit multiplier should be loaded from memory as few times as possible to compute partial products for its rows, and the columns should be accumulated individually in registers. By scheduling multiplications of the bits of the multiplier as close together as possible, the algorithm stems the growth of the number of registers needed to accumulate the final result while preventing repeated accesses to memory for each bit.

The Hybrid Squaring algorithm proceeds as the standard multiplication algorithm, but it additionally takes advantage of the fact that in a squaring, each partial product appears twice.

C. Projective Coordinates

As discussed in the handbook chapter by Doche and Lang [4] and Hankerson, Menezes, and Vanstone [5], points on an elliptic curve can be represented in the form (x, y, z) as projective coordinates rather than affine coordinates in (x, y). Representing points in this way helps improve the speed of point addition by replacing modular inversion with a small number of additions and multiplications. For the sake of space because the formulas are cumbersome (and because these were covered in a homework), we defer to [4] and [5] to give algorithms for point addition and multiplication using projective coordinates. However, we note [5] explain that using projective coordinates, a point doubling in Jacobian projective coordinates can be computed using only six fields squarings and four field multiplications. [4] gives an algorithm for computing point addition using twelve field multiplications and four squarings.

Although computation on projective coordinates is generally faster than affine coordinates, it is not sufficient to only perform computation on projective coordinates. Sometimes it is useful to add a point in projective coordinates with a point in affine coordinates without needing to convert between coordinate systems. Doche and Lang [4] show that if one of the points has a z value of 1, then it reduces to eight field multiplications and three field squarings, which is possible with a trivial conversion of the affine point to a Jacobian projective point. However, the algorithm requires storage for nine intermediate values.

In addition, Hankerson, Menezes, and Vanstone [5] show that if computing repeated point doublings in Jacobian projective coordinates, it might be faster to use a different algorithm than repeated application of the doubling formula. When m consecutive doublings are computed, the algorithm (which we defer to [5]) trades $m - 1$ field additions, $m - 1$ divisions by two, and a multiplication for only two field squarings.

D. Sliding Window for Scalar Multiplication in NAF

Rather than represent a number in $GF(p)$ (or $GF(2^m)$) strictly in its binary representation, it is useful to represent (or recode) the number in Non-Adjacent Form (NAF), where digits are drawn from the set $(0, 1, -1)$, and the representation is analogous to the binary, except that a negative number indicates subtraction of that power of the base. The advantages of this representation include the fact that it is guaranteed for no non-zero digits to border each other, allowing for fewer maximum operations in any algorithm that depends on non-zero digits to decide a step.

An efficient algorithm for scalar multiplication of a number in $GF(p)$, given by [5], is presented in Algorithm 2.

Input: Window width w, positive integer k, $P \in E(\mathcal{F}_q)$
Output: kP
Compute $NAF(k) = \sum_{i=0}^{l-1} k_i 2^i$
Compute $P_i = iP$ for $i \in \{1, 3, 5, ..., 2^w - 1\}$
$Q \leftarrow \infty$
for $i = l - 1 \rightarrow 0$ **do**
    $Q \leftarrow 2Q$
    if $k_i \neq 0$ **then**
        if $k > 0$ **then**
            $Q \leftarrow Q + P_{k_i}$
        else
            $Q \leftarrow Q - P_{-k_i}$
        end if
    end if
end for
return Q

Algorithm 2: Window Non-Adjacent Form Method for Point Multiplication

The above textbook-like multiplication algorithm scans bits of the multiplier from most significant to least significant determine when to compute a point doubling and addition. The approach achieves additional performance benefits by precomputing the table of small scalar multiples of P and using them for additions. However, this approach can be further improved using a "sliding window" trick, given by [5], where instead, k bits of input are viewed at a time rather than carrying out a bitwise multiplication, or as in this algorithm, additions for every bit flag.

Input: Window width w, positive integer k, $P \in E(\mathcal{F}_q)$
Output: kP
Compute $NAF(k) = \sum_{i=0}^{l-1} k_i 2^i$
Compute $P_i = iP$ for $i \in \{1, 3, 5, ..., 2(2^w - (-1)^w)/3 - 1\}$
$Q \leftarrow \infty$, $i \leftarrow l - 1$
while $i \geq 0$ **do**
  if $k_i = 0$ **then**
    $t \leftarrow 1$, $u \leftarrow 0$
  else
    Find largest $t \leq w$ such that $u \leftarrow (k_i, ..., k_{i-t+1})$ is odd
  end if
  $Q \leftarrow 2^t Q$
  if $u > 0$ **then**
    $Q \leftarrow Q + P_u$
  else if $u < 0$ **then**
    $Q \leftarrow Q - P_{-u}$
  end if
  $i \leftarrow i - t$
end while
return Q

Algorithm 3: Sliding Window Method for Point Multiplication

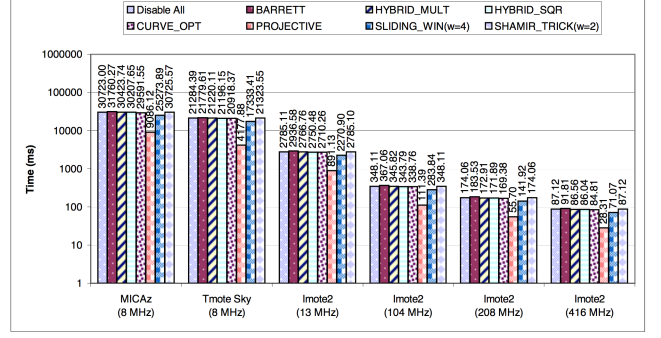
The "sliding window" algorithm is given in Algorithm 3. The algorithm takes advantage of the NAF to "look ahead" as far as the window size w to find the next power of 2 by which to multiply Q (just 1 if k is 0, and there is no point addition) and then looks up in the precomputed table which point to add (or subtract).

E. Shamir's Trick

Shamir's trick is used only for verification of ECDSA signatures, so we will not pay much attention to it here. However, the trick allows computation of the form $aP + bQ$ for close to the cost of a single scalar multiplication rather than two multiplications an an addition. The algorithm is similar to the sliding window trick, where for every bit of a and b the algorithm checks whether to add P or Q or $P + Q$ to an accumulated value that is doubled at every bit after initialization at ∞, rather than just to add or subtract P as in Algorithm 2.

F. Pseudo-Mersenne Primes

A Mersenne Prime is of the form $p = 2^n - 1$; a pseudo-Mersenne prime is of the form $2^n - c$ for some $c << 2^n$. When constructing a finite field over which to establish an elliptic curve, it is advantageous to choose a pseudo-Mersenne prime because reductions modulo the prime can be computed with a few modular multiplications and additions without the need for any divisions. Indeed, some of the prime fields recommended by NIST use pseudo-Mersenne primes.



(c) Sig. generation time when all other optimizations are disabled (case A)

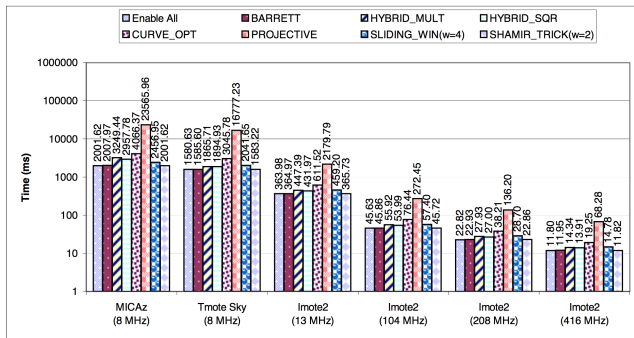
Fig. 1. Signature Generation Times with One Optimization Enabled [1]

III. Evaluation

The authors of TinyECC evaluated the performance of each optimization they employed by running a series of experiments that they stratified into two categories. First, they configured and ran the code with only the optimization they wanted to test enabled; this was considered case A. Second, they configured and ran the code with all of the optimizations enabled except for the one they were testing; this was considered case B. The result was a large set of tests in which no optimizations were present, only one optimization was present at a time, all optimizations except for one were present, and all optimizations were present, allowing for thorough analysis of the effects of each optimization in comparison to the others. While graphs were constructed for initialization, signature generation, and signature verification times, we only reproduce here the graphs constructed for signature generation, since we consider initialization to be a one-time cost and verification results are similar to generation with the exception of Shamir's trick, which improves the speed of verification. The tables are given in figures 1 and 2.

It was clear that the most useful technique was the use of projective coordinates. The least useful trick, surprisingly, is the use of Barrett reductions for modular multiplications. It should not be surprising that using projective coordinates improves the speed (and power consumption) most of all of the optimizations, since the methods specifically target the most expensive elliptic curve computations and trade them for more operations of more efficient nature.

However, it is very surprising that Barrett Reductions did not strongly improve the performance of signature generation and even slowed it down on every architecture when it was the only optimization enabled. This slow-down cannot be explained as an issue in implementation, where every Barrett Reduction constant had to be recomputed before computing the modulus, since (although not shown here), Barrett reductions were shown to contribute heavily to initialization times.



(d) Sig. generation time when all other optimizations are enabled (case B)

Fig. 2. Signature Generation Times with All But One Optimization Enabled [1]

In addition, the TinyECC paper [1] very unsatisfactorily does not contribute ideas about why Barrett Reduction has this effect. Indeed, one would assume that with modular reductions within the prime field necessary for almost every ECC operation that requires multiplication or squaring, it would have similar impact to projective coordinates by eliminating divisions! It is possible that the reason for Barrett Reduction not having a very large effect while other optimizations are enabled is that curve-specific optimizations already achieve most of the gain. When using pseudo-Mersenne primes as the size of the field, it is already possible to achieve very efficient modular reductions, obviating the need for another efficient modular reduction algorithm. This is supported by the fact that the curve-specific optimizations in the chart, which are simply choosing pseudo-Mersenne primes, have the second biggest effect on the execution time. However, this does not explain why Barrett Reduction *increases* the execution time when it is the only "optimization" enabled.

The given results, which are scaled logarithmically in terms of time, highlight the effectiveness of projective coordinates. In each case, not using projective coordinates results in a slowdown of at least three times for signature generation! This information highlights the value of eliminating highly expensive field inversions and replacing them with multiplications and additions, even modular multiplications. It is nearly possible to say without qualification that converting to projective coordinates to use more efficient algorithms is by far worth the cost of keeping a small number of extra temporary variables to hold intermediate computations.

IV. Conclusion

We see that for TinyECC, efficient computation of elliptic curve cryptography focuses greatly on methods to manipulate the representations of numbers to find efficient processes for point additions and multiplications, often trading expensive inversions for multiplications and additions or building multiplications from additions and scalar multiplications. Of all the optimizations that TinyECC implemented, the use of projective coordinates to reduce expensive field inversions at the expense of squarings and multiplications was the most effective. Next most effective was choosing an effective prime to construct the field so that modular reductions were highly efficient, mitigating the possibly prohibitive costs of modular multiplications.

Although TinyECC does not specifically mention considerations for computation on elliptic curves in $GF(2^m)$ it supports arithmetic in that field. However, it does not mention support for conversions in $GF(2^m)$ between polynomial representation and normal basis representation; if the conversions between point representations are efficient, they may achieve even better power performance for TinyECC on devices over wireless sensor networks by achieving squarings for the price of a bitwise shift.

References

[1] An Liu and Peng Ning, "Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks," in *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on*. IEEE, 2008, pp. 245–256.

[2] "Tiny ecc source home page," http://discovery.csc.ncsu.edu/software/TinyECC/.

[3] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz, "Comparing elliptic curve cryptography and rsa on 8-bit cpus," in *Cryptographic hardware and embedded systems-CHES 2004*, pp. 119–132. Springer, 2004.

[4] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren, *Handbook of elliptic and hyperelliptic curve cryptography*, CRC press, 2005.

[5] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone, *Guide to elliptic curve cryptography*, Springer Science & Business Media, 2006.