TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks

An Liu, Peng Ning
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
Email: {aliu3, pning}@ncsu.edu

Abstract

Public Key Cryptography (PKC) has been the enabling technology underlying many security services and protocols in traditional networks such as the Internet. In the context of wireless sensor networks, elliptic curve cryptography (ECC), one of the most efficient types of PKC, is being investigated to provide PKC support in sensor network applications so that the existing PKC-based solutions can be exploited.

This paper presents the design, implementation, and evaluation of TinyECC, a configurable *library for ECC operations in wireless sensor networks. The primary objective of TinyECC is to provide a* ready-to-use, publicly available *software package for ECC-based PKC operations that can be* flexibly configured and integrated *into sensor network applications. TinyECC provides a number of optimization switches, which can turn specific optimizations on or off based on developers' needs. Different combinations of the optimizations have different execution time and resource consumptions, giving developers great flexibility in integrating TinyECC into sensor network applications. This paper also reports the experimental evaluation of TinyECC on several common sensor platforms, including MICAz, TelosB, Tmote Sky, and Imote2. The evaluation results show the impacts of individual optimizations on the execution time and resource consumptions, and give the most computationally efficient and the most storage efficient configuration of TinyECC.*

1. Introduction

Recent technological advances have made it possible to develop wireless sensor networks consisting of a large number of low-cost, low-power, and multi-functional sensor nodes that communicate over short distances through wireless links [8]. Such sensor networks are ideal candidates for a wide range of applications such as monitoring of critical infrastructures, data acquisition in hazardous environments, and military operations. The desirable features of wireless sensor networks have attracted many researchers to develop protocols and algorithms that can fulfill the requirements of these applications.

Security services such as authentication and key management are critical to communication security in wireless sensor networks as well as the security of sensor network applications. In traditional networks such as the Internet, Public Key Cryptography (PKC) has been the enabling technology underlying many security services and protocols (e.g., SSL [4] and IPsec [21, 22]). However, in wireless sensor networks, PKC has not been widely adopted due to the resource constraints on sensor platforms, particularly the limited and depleteable battery power.

There has been intensive research aimed at developing techniques that can bypass PKC operations in sensor network applications. For example, there has been a substantial amount of research on random key pre-distribution for pairwise key establishment (e.g., [12, 16, 17, 27, 29]) and broadcast authentication (e.g., [28, 30, 38]). However, these alternative approaches do not offer the same degree of security or functionality as PKC. For instance, none of the random key pre-distribution schemes can guarantee key establishment between any two nodes and tolerate arbitrary node compromises at the same time. As another example, the aforementioned broadcast authentication schemes, which are all based on TESLA [37], require loose time synchronization, which itself is a challenging task to achieve in wireless sensor networks. In contrast, PKC can address all these problems easily. Pairwise key establishment can always be achieved using, for example, the Diffie-Hellman (DH) key exchange protocol [15], without suffering from the node compromise problem. Similarly, broadcast authentication can be provided with, for example, the ECDSA digital signature scheme [9], without requiring time synchronization. Thus, it is desirable to explore the application of PKC on resource constrained sensor platforms.

There have been a few recent attempts to use PKC in wireless sensor networks [19, 26, 31, 40], which demonstrate that it is feasible to perform limited PKC operations on the current sensor platforms such as MICAz motes [2]. Elliptic Curve Cryptography (ECC) has been the top choice among various PKC options due to its fast computation, small key size, and compact signatures. For example, to provide equivalent security to 1024-bit RSA, an ECC scheme only needs 160 bits on various parameters, such as 160-bit finite field operations and 160-bit key size [10].

Despite the recent progress on ECC implementations on sensor platforms, all the previous attempts [19, 31, 40] have limitations. In particular, all these attempts were developed as independent packages/applications without seriously considering the resource demands of sensor network applications. As a result, developers may found it difficult, and sometimes impossible, to integrate an ECC implementation with the sensor network applications, though the ECC implementation may be okay on its own. For example, an ECC implementation may require so much RAM that it is impossible to fit both the sensor network application and the ECC implementation on the same node.

Moreover, various optimization techniques are available to speed up the ECC operations. Such optimizations, however, typically will increase the ROM and RAM consumptions, though they may reduce the execution time and energy consumption. It is not clear what optimizations should be used and how they should be combined to achieve the best trade-off among security protection, computation overheads, and storage requirements. Additional research is necessary to clarify these issues and facilitate the adoption of ECC-based PKC in wireless sensor networks.

In this paper, we present the design, implementation, and evaluation of TinyECC, a *configurable* library for ECC operations in wireless sensor networks.[1] The primary objective of TinyECC is to provide a *ready-to-use, publicly available* software package for ECC-based PKC operations that can be *flexibly configured and integrated* into sensor network applications.

Targeted at TinyOS [6], TinyECC is written in nesC [18], with occasional in-line assembly code to achieve further speedup for popular sensor platforms including MICAz [2], TelosB [5], Tmote Sky [7], and Imote2 [1]. A unique feature of TinyECC is its *configurability*. TinyECC includes almost all known optimizations for ECC operations. Each optimization is controlled by a software switch, which can turn the optimization on or off based on developers' need. Different combinations of optimizations have different ROM/RAM consumptions, execution time, and energy consumption. This gives the developers great flexibility in integrating TinyECC in their applications.

To provide guidance in using TinyECC, we perform a series of experiments with different combinations of activated optimizations. To understand the impact of each optimization technique, we compare the execution time, ROM/RAM consumptions, and energy consumptions with and without the given optimization enabled on MICAz [2], TelosB [5], Tmote Sky [7], and Imote2 [1]. In addition, our experiments also present the performance results and the resource usages for the most computationally efficient configuration (i.e., fastest execution and least energy consumption) and the most storage-efficient configuration (i.e., least ROM and RAM usage) of TinyECC on these common sensor platforms, respectively.

The contribution of this paper is two-fold: First, we develop TinyECC, a configurable library for ECC operations in wireless sensor networks, which allows flexible integration of ECC-based PKC in sensor network applications. Second, we perform a substantial amount of experimental evaluation using representative sensor platforms, including MICAz [2], TelosB [5], Tmote Sky [7], and Imote2 [1]. The experimental results provide useful experience and guidance for developers to choose different TinyECC optimizations for their needs.

The remainder of this paper is organized as follows. Section 2 discusses the design principles of TinyECC. Section 4 describes the optimization techniques adopted by TinyECC. Section 5 discusses the implementation of TinyECC. Section 6 presents the experimental evaluation of TinyECC on MICAz, TelosB, Tmote Sky, and Imote2. Section 7 discusses the related work, and Section 8 concludes this papers and points out some future research directions.

2. Design Principles

As mentioned earlier, the primary objective of TinyECC is to provide a *ready-to-use, publicly available* software package for ECC-based PKC operations that can be *flexibly configured and integrated* into sensor network applications. To make sure we achieve this objective, we follow several principles in the design and development of TinyECC.

Security: TinyECC should provide PKC schemes that have proved to be secure. To follow this principle, TinyECC only includes support for the well-studied ECC schemes such as ECDSA, ECDH, and ECIES, which are defined in the Standards for Efficient Cryptography [10]. Moreover, TinyECC also includes elliptic curve parameters recommended by SECG (Stands for Efficient Cryptography Group), such as `secp160k1`, `secp160r1` and `secp160r2`, as defined in [11].

Portability: TinyECC should run on as many sensor platforms as possible. Due to this reason, we choose to implement TinyECC on TinyOS [6], which is a popular, open-source OS for networked sensors. All the TinyECC components have nesC [18] implementations, though some modules also include inline assembly code, which can be turned on for faster execution on some sensor platforms. This allows TinyECC to be compiled and used on any sensor platform that can run TinyOS. TinyECC has been tested successfully on MICAz,

[1] TinyECC 1.0 and its previous versions are publicly available at `http://discovery.csc.ncsu.edu/software/TinyECC/`.

TelosB, Tmote Sky, and Imote2.

Resource Awareness and Configurability: TinyECC should accommodate the typical resource constraints on sensor nodes. Moreover, TinyECC should allow flexible configuration so that it can take advantage of the available resources on a wide spectrum of sensor platforms. To follow this principle, TinyECC is implemented carefully to avoid unnecessary resource usage. Moreover, TinyECC uses a set of optimization switches, which can be turned on or off to achieve different combinations of performance and resource consumptions.

Efficiency: TinyECC should be computationally efficient to reduce the battery consumption as well as the delay introduced by PKC operations. We make three design decisions to improve the efficiency of TinyECC. The first is about the type of finite fields over which the ECC operations are performed. ECC can be implemented over either a prime field F_p, where p is a large prime number, or a binary extension field F_{2^m}, where m is an integer. Since arithmetic operations over F_{2^m} are insufficiently supported by microprocessors, we choose to support prime fields F_p in TinyECC. Second, we adopt almost all existing optimizations for ECC operations in TinyECC. As mentioned earlier, these optimizations can be turned on or off to balance the efficiency and the resource requirements. Third, we include inline assembly code in critical parts of TinyECC for popular sensor platforms, including MICAz, TelosB, Tmote Sky, and Imote2.

Functionality: TinyECC should support the typical demands for PKC. To follow this principle, the current version of TinyECC includes a digital signature scheme (ECDSA), a key exchange protocol (ECDH), and a public key encryption scheme (ECIES). These cover all typical uses of PKC.

3. Background on ECC

In this section, we give a brief introduction to ECC. The reader is referred to [20, 41] for more details.

Elliptic curve cryptography (ECC) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields [41]. Elliptic curves used in cryptography are typically defined over two types of finite fields: prime fields F_p, where p is a large prime number, and binary extension fields F_{2^m}. For space reasons, we focus on elliptic curves over F_p in this paper.

An elliptic curve over F_p is defined by a cubic equation $y^2 = x^3 + ax + b$, where $a, b \in F_p$ are constants such that $4a^3 + 27b^3 \neq 0$ [20, 41]. An elliptic curve over F_p consists of the set of all pairs of affine coordinates (x, y) for $x, y \in F_q$ that satisfy an equation of the above form and an infinity point $\mathcal{O}$. The points on an elliptic curve form an abelian group with $\mathcal{O}$ as the additive identity. (The formulas defining point addition and its special case, point doubling, can be found in [20, 41].)

For any point G on an elliptic curve, the set $\{\mathcal{O}, G, 2G, 3G, ...\}$ is a cyclic group [20, 41]. The calculation of kG, where k is an integer, is called a *scalar multiplication*. The problem of finding k given points kG and G is called the *elliptic curve discrete logarithm problem (ECDLP)*. It is computationally infeasible to solve ECDLP for appropriate parameters [20, 41]. The hardness of ECDLP allows several cryptographic schemes based on elliptic curves.

TinyECC includes three well-known ECC schemes: (1) the Elliptic Curve Diffie-Hellman (ECDH) key agreement scheme, (2) the Elliptic Curve Digital Signature Algorithm (ECDSA), and (3) the Elliptic Curve Integrated Encryption Scheme (ECIES). ECDH is a variant of the Diffie-Hellman key agreement protocol [15] on elliptic curve groups. ECDSA is a variant of the Digital Signature Algorithm (DSA) [35] that operates on elliptic curve groups. ECIES is a public-key encryption scheme which provides semantic security against an adversary who is allowed to use chosen-plaintext and chosen-ciphertext attacks [41]. ECIES is also known as the Elliptic Curve Augmented Encryption Scheme (ECAES) or simply the Elliptic Curve Encryption Scheme. These ECC schemes allow smaller key sizes for similar security level to the alternatives such as the original DH and DSA schemes. For each of the schemes, a party that would like to use the scheme needs to agree on some domain parameters such as the elliptic curve and a point G on the curve, and must have a key pair consisting of a private key d and a public key $Q = dG$. The specification of ECDH, ECDSA, and ECIES can be found in [10, 20].

4. Optimizations Adopted by TinyECC

In this section, we briefly discuss the optimization techniques adopted by TinyECC. We will omit the details, since the focus of this paper is not these individual optimization techniques. More information about these techniques can be found in the relevant references.

4.1. Optimizations for Large Integer Operations

Barrett Reduction [33]: The most straightforward way to perform large integer modular reductions is to use division [23]. A nice side effect of such a method is that it reuses the code of division, thus resulting in compact code size.

Barrett Reduction is an alternative method for modular reduction [33]. It converts the reduction modulo an arbitrary integer to two multiplications and a few reductions modulo integers of the form 2^n. When used to reduce a single number, Barrett reduction is slower than a normal division algorithm. However, when used to reduce various numbers modulo the same number many times, by pre-computing some value, Barrett reduction can achieve faster speed than modular reductions obtained by division. Details of Barrett reduction can be found in [33].

In TinyECC, since almost all the modular operations are modulo the same prime number p, Barrett reduction can potentially speed up the computation. However, this requires the implementation of a separate reduction algorithm, which implies larger code size (i.e., more ROM requirement) on sensor nodes. In addition, Barrett reduction also increases the RAM consumption. Assume the target microprocessor has a w-bit word size. Given a finite field F_p, where p is a k words long prime number, Barrett reduction requires the pre-computation of $\mu = \lfloor \frac{b^k}{p} \rfloor$, where $b = 2^w$ (e.g., $b = 2^8$ on a 8-bit processor). This number μ has to be stored and used throughout all the modular reductions. Thus, to exchange for faster computation, Barrett reduction requires more ROM and RAM than the traditional division-based modular reduction.

Hybrid Multiplication and Hybrid Squaring [19]: Standard large integer multiplication algorithms [23] store the operands and the product in arrays. When such an algorithm is implemented in a high-level language such as nesC, the compiler cannot use the registers in the microprocessor efficiently, and the binary code usually needs to load the operands from memory to registers multiple times [19]. Gura et al. [19] proposed a hybrid multiplication algorithm, which was intended for assembly code. This algorithm can maximize the utilization of registers and reduce the number of memory operations. TinyECC adopts this hybrid multiplication algorithm for MICAz [2], TelosB [5]/Tmote Sky [7], and Imote2 [1]. Indeed, the code can be used on any sensor platforms that have processors using the same instruction sets. The implementation of hybrid multiplication has width 4 or 5 for MICAz, depending on the curve parameters, and has width 1 for TelosB/Tmote Sky and Imote2 due to the small number of registers on them.

In addition to hybrid multiplication, we also customize the hybrid multiplication algorithm for squaring operations by using the fact that the two multiplicative operands in squaring are the same. This further reduces the execution time for squaring at the cost of larger code size.

4.2. Optimizations for ECC Operations

Projective Coordinate Systems [20]: As discussed earlier, an elliptic curve consists of the infinity point $\mathcal{O}$ and the set of points in the affine coordinates (x,y) for $x,y \in F_p$ that satisfy the defining equation. Alternatively, a point on an elliptic curve can be represented in a projective coordinate system in the form of (x,y,z).

Point addition and point doubling are critical operations in ECC, which are buidling blocks for scalar multiplications required by all ECC schemes. These operations in affine coordinate system require modular inversion operations, which are much more expensive than other operations such as modular multiplications. Using a projective coordinate system [20], modular inversions can be removed with the compensation of a few modular multiplications. As a result, the execution times of point addition and point doubling based on projective coordinate system are faster than those based on affine coordinate system, respectively [20].

TinyECC uses two additional optimizations along with projective coordinate representation, which can further reduce both the execution time and the program size. The first is a *mixed point addition algorithm* [20], which adds a point in projective coordinate and a second point in affine coordinate. This algorithm can be used in scalar multiplications to further reduce the number of modular multiplications and squares, leading to smaller and faster code. The second is *repeated Doubling* [20] for scalar multiplication. If consecutive point doublings are to be performed, the repeated doubling algorithm may be used to achieve faster performance than repeated use of the doubling formula. In m consecutive doublings, this algorithm trades $m-1$ field additions, $m-1$ divisions by two, and a multiplication for two field squarings (in comparison with repeated applications of the plain point doubling algorithm) [20].

Though reducing the execution time, the projective coordinate representation requires larger code size (for more complex formula) and more RAM (for storing additional variables) than the affine coordinate system.

Sliding Window for Scalar Multiplications [20]: Scalar multiplication is a basic operation used by all ECC schemes. It is in the form of kP, where k is an integer and P is a point on an elliptic curve. In the most straightforward method to compute kP, k is scanned from the most significant bit to the least significant bit. When each bit is scanned, the algorithm needs to compute a point doubling. When the scanned bit is "1", the algorithm also needs to perform a point addition. The sliding window method can speed up the scalar multiplication by scanning w bits at a time. Each time when a w-bit window is scanned, the algorithm needs to perform w point doublings. By precomputing $2P$, $3P$, ..., and $(2^w-1)P$, the sliding window method only needs to perform 1 point addition every w bits, and thus has less computational cost.

It is easy to see that the sliding window method will increase both the ROM (for additional code size) and RAM (for storing the pre-computed points) consumptions.

Shamir's Trick [20]: This optimization is only used for the verification of ECDSA signatures. The verification of ECDSA signature requires the computation of the form $aP + bQ$, where a,b are integers and P,Q are two points on an elliptic curve. A straightforward implementation requires two scalar multiplications and a point addition. However, Shamir's trick allows us to compute the above value at a cost close to one scalar multiplication. Specifically, with pre-computed $P+Q$, we may scan the (same) bits of a and b from the most significant one to the least significant one. For each bit, we need double the intermediate value, which is initialized as the infinity point. If the scanned bit positions are $\langle a_i = 0, b_i = 1\rangle$, $\langle a_i = 1, b_i = 0\rangle$, or $\langle a_i = 1, b_i = 1\rangle$, we add P, Q, or $P+Q$ to the intermediate value. This reduces two scalar multiplications to a bit more expensive than one such operation.

Similar to the sliding window method, Shamir's trick will increase both the ROM (for additional code size) and RAM (for storing the pre-computed $P+Q$) consumptions.

Curve Specific Optimization [19]: A number of elliptic curves specified by NIST [36] and SECG [11] use pseudo-Mersenne primes. A pseudo-Mersenne prime is of the form $p = 2^n - c$, where $c \ll 2^n$. Reduction modulo a pseudo-Mersenne prime can be performed by a few modular multiplications and additions without any division operation. As a result, the time for modular reduction can be reduced significantly. Thus, using elliptic curves over a pseudo-Mersenne prime can achieve additional performance gain.

5. Implementation

We implemented TinyECC on TinyOS [6], an open source operating system designed for wireless embedded sensor networks. The current version of TinyECC provides support for ECDSA (digital signatures), ECDH (pairwise key establishment), and ECIES (PKC-based encryption). Most of the code were written in nesC [18] for portability reasons. To best harness the capabilities of the processors on the popular sensor platforms such as MICAz and TelosB, we also provided inline assembly implementation of some critical operations, such as large integer multiplications.

To save implementation efforts, we ported the C code of large integer operations in RSAREF 2.0 [24] to nesC code on TinyOS. These include modular addition, subtraction, multiplication, division, inverse, and exponentiation operations. We then implemented all the elliptic curve operations and the optimization techniques discussed earlier.

TinyECC has been released publicly at `http://discovery.csc.ncsu.edu/software/TinyECC/`. Some preliminary versions have been adopted by other researchers (e.g., [14, 25, 32]). As discussed earlier, starting from the current version, we added a set of optimization switches to provide flexible configuration of TinyECC so that it can be integrated into sensor applications with different resource consumptions and performance demands.

Table 1 lists the optimization switches available in the current version of TinyECC. Most optimization switches can be turned on or off by a simple configuration at compile time, or slight modification in the source code. A few optimization switches requires additional care. Specifically, for hybrid multiplication and squaring techniques, a macro indicating specific hardware platform (e.g., MICAz, Imote2) should be defined, so that TinyECC will use the inline assembly code with the right instruction set. Moreover, when the sliding window method is used, an additional parameter defining the size of the window (e.g., $w = 4$) must be defined. Finally, curve specific optimizations only work for pseudo-Mersenne primes. Thus, when curve specific optimization is enabled, a prime number in the appropriate form must be defined as well.

6. Evaluation

We performed a series of experiments to evaluate TinyECC on four representative sensor platforms, including MICAz [2], TelosB [5], Tmote Sky [7], and Imote2 [1].

The objective of these experiments is three-fold: First, we would like to measure the performance and resource consumption of TinyECC on a spectrum of sensor platforms, ranging from the low-end ones (such as MICAz, TelosB, and Tmote Sky) to high-end ones (such as Imote2). Second, we would like to understand the impact of the optimizations adopted by TinyECC on the performance and resource consumption. Finally, we would like to provide detailed performance results and resource demands for the most commonly desirable configurations, including the configuration that provides the fastest execution time and the configuration that requires the least memory consumption. The former has the least energy consumption, while the latter is the easiest one to integrate into sensor applications.

6.1. Methodology and Experiment Setup

Evaluation Methodology: Given seven optimization switches, four sensor platforms, where Imote2 has multiple CPU frequencies due to dynamic voltage scaling, many possible elliptic curves, and three ECC-based PKC schemes, there are a large number of experiments to perform if we have to observe the differences in performance and resource consumptions in all cases.

To simplify the scenarios, we adopt the following methodology in our experiments. For each optimization switch, we perform two sets of experiments, referred to as *case A* and *case B*, respectively. In case A, we disable all the other optimizations, and then obtain the performance and resource consumption metrics when the given optimization is enabled and disabled, respectively. In case B, we enable all the other optimizations and obtain the evaluation metrics again when the given optimization is enabled and disabled, respectively. The differences in these metrics reflect the impact of the given optimization technique.

Moreover, as discussed earlier, we also perform additional experiments to examine in detail two commonly desirable configurations: the one that provides the fastest execution time, and the one that requires the least storage.

Experiment Setup: We evaluate TinyECC on the latest CVS version of TinyOS 1.x [6]. As discussed earlier, we choose four representative sensor platforms, MICAz, TelosB, Tmote Sky, and Imote2, for the experiments, since they are popular sensor platforms and cover the 8-bit, 16-bit and 32-bit processors. Other sensor platforms (e.g., Mica2, Mica2Dot) are expected to perform similarly to one of these platforms,

Table 1. TinyECC Optimization Switches

Method	Optimization Switch	Category	Description
Barrett Reduction	BARRETT	large number	Turn this switch on to allow Barrett reduction.
Hybrid Multiplication	HYBRID_MULT	large number	Turn this switch on to allow hybrid multiplication in inline assembly.
Hybrid Squaring	HYBRID_SQR	large number	Turn this switch on to allow hybrid squaring in inline assembly.
Projective Coordinate System	PROJECTIVE	EC	Turn this switch on to use projective coordinate system along with mixed point addition and repeated doubling.
Sliding Window Method	SLIDING_WIN	EC	Turn this switch on to use sliding window method for scalar multiplication. A window size (e.g. $w = 4$) has to be defined along with this switch.
Shamir's Trick	SHAMIR_TRICK	EC	Turn this switch on to allow Shamir's trick when verifying ECDSA signatures. A window size (e.g. $w = 2$) has to be defined along with this switch.
Curve-Specific Optimization	CURVE_OPT	EC	Turn this switch on to allow curve specification optimization. This has to be used for the curves defined over pseudo-Mersenne primes [11, 36].

due to their adoption of the same processors.

TelosB and Tmote Sky have (almost) the same hardware. The only difference is that TelosB can only run at 4 MHz, while Tmote Sky can run at 8 MHz when an external resistor is enabled. We configure Tmote Sky to run at 8 MHz in our experiments. As a high-end sensor platform, Imote2 uses an XScale processor and supports dynamic voltage scaling. To obtain a relatively complete view of Imote2, we use four different frequencies on Imote2 in our experiments: 13MHz, 104MHz, 208MHz, and 416MHz.

By default, TinyECC includes all 128-bit, 160-bit and 192-bit ECC parameters recommended by SECG [11]. It is well-known that 160-bit ECC has the same security level as 1024-bit RSA. We selected a 160-bit elliptic curve `secp160r1` [11] to evaluate the impact of individual optimization techniques. Note that the actual selection of curves depends on the security needs in the sensor network applications, and is outside of the scope of this paper.

We used the following evaluation metrics in all experiments: ROM consumption (byte), RAM consumption (byte), execution time (ms), and energy consumption (millijoule). We used the `check_size.pl` script in the TinyOS distribution to obtain the ROM and RAM sizes required by the TinyECC components. The execution time was measured directly on the sensor nodes. To get the overall performance result, we randomly generated the parameters other than those defining the curves (e.g., random message, random public and private key pairs), and obtained the execution time for each data point by taking the average of 10 test instances. The energy consumption was then calculated as $U \times I \times t$ based on the execution time (t), the voltage (U), and current draw (I) on these sensor platforms [1, 2, 5, 7].

6.2. Evaluation Results

6.2.1. Impact of Individual Optimizations

Public Key Generation: We first present the impact of individual optimizations on the execution time of public key generation in ECDSA, ECIES, and ECDH, as shown in Figure 1.

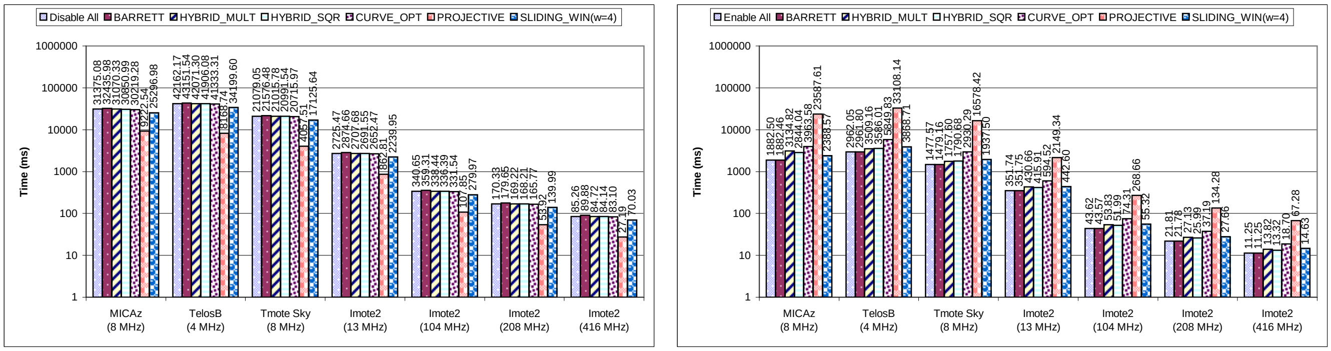
Since *SHAMIR_TRICK* is only for ECDSA signature verification, it has no effect on public key generation. Thus, we skip *SHAMIR_TRICK* in Figure 1. From Figure 1, we can see that *PROJECTIVE* is the most effective switch in both case A and B. In case A, *SLIDING_WIN* is the second most effective switch, while in case B, *CURVE_OPT* is the second most effective switch. In both cases, *HYBRID_MULT* and *HYBRID_SQR* have similar effects.

Barratt Reduction: We notice that *BARRETT* is not as effective as expected. The public key generation is even slower in case A when *BARRETT* is enabled, as Figure 1(a) shows. Barrett reduction requires 2 large number multiplications and 4 large number division (replaced by memory shifting). Since memory operation is slow for low-end sensor platforms, the Barrett reduction is not necessarily faster than division in TinyECC when *HYBRID_MULT* are disabled.

	Division	Barrett reduction w/o *HYBRID_MULT*	Barrett reduction w/ *HYBRID_MULT*
MICAz (8 MHz)	221.07	394.59	169.79
TelosB (4 MHz)	196.50	367.68	243.04
Tmote Sky (8 MHz)	98.82	184.5	121.86
Imote2 (13 MHz)	71.29	161.28	64.80
Imote2 (104 MHz)	8.97	20.22	8.16
Imote2 (208 MHz)	4.52	10.14	4.11
Imote2 (416 MHz)	2.39	5.22	2.18

Table 2. Execution time (ms) for modular reduction through division and Barrett reduction

To gain more insights into this issue, we perform additional tests on the execution time of normalized division and barrett reduction (with and without *HYBRID_MULT*) by randomly generating a 320-bit large number and computing mod with the modular p defined in `secp160r1` for 100 rounds. The results are given in Table 2. These results indicate that Barrett reduction is slower than normalized division when *HYBRID_MULT* is disabled, but faster when *HYBRID_MULT* is enabled for MICAz and Imote2. In other words, the Barrett reduction optimization should be used along with *HYBRID_MULT* to be helpful for these two platforms. Since the hardware multiplier of TelosB/Tmote Sky is not part of MSP430 CPU, the use of this hardware multiplier involves loading and reading peripheral registers. The Barrett reduction is slower than normalized division for TelosB/Tmote Sky



(a) Public key gen time when all other optimizations are disabled (case A)

(b) Public key gen time when all other optimizations are enabled (case B)

Figure 1. Execution time of public key generation

with hybrid multiplication.

ECDSA: Now we present the impacts of individual optimizations on the execution time of ECDSA. There are three aspects of the execution time. Figures 2(a) and 2(b) show the initialization time required to prepare for ECDSA in cases A and B, respectively. Figures 2(c) and 2(d) show the signature generation time in cases A and B, respectively. Figures 2(e) and 2(f) show the signature verification time in cases A and B, respectively.

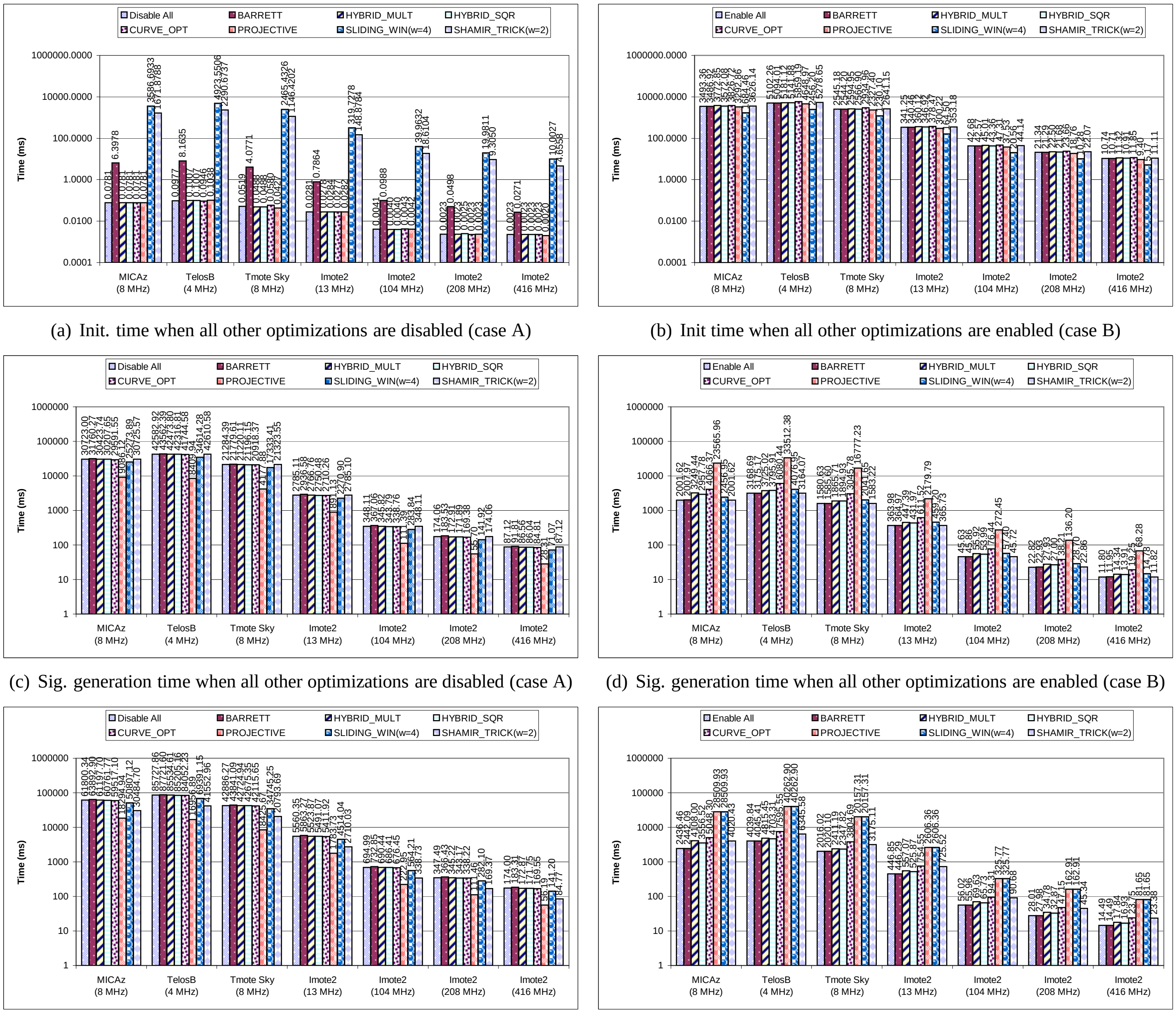
In the initialization of ECDSA, TinyECC needs to precompute μ for Barrett reduction, a few points for the sliding window method, and a few points for Shamir's trick. In case A, as Figure 2(a) shows, only these 3 optimization techniques have impact on the initialization time. For MICAz, the initialization of the sliding window method with window size 4 requires 3,587 ms, which is longer than Shamir's trick (1,672 ms for window size 2) and barrett reduction (6 ms). The same situation applies to TelosB, Tmote Sky, and Imote2. If we disable all these three techniques, the initialization time of ECDSA is close to 0. In case B, as Figure 2(b) shows, the disabling of selected optimization technique doesn't reduce the initialization time dramatically. Only the disabling of the sliding window method can reduce the initialization time to half.

In Figure 2, we can see that *PROJECTIVE* is the most effective switch to improve the speed of signature generation and verification. In case A, by enabling the *PROJECTIVE* switch, the signature generation and verification of all platforms can speed up at least 3 times. In case B, if we disable the *PROJECTIVE* switch, the signature generation and verification has at least 6 times slowdown compared with enabling all optimization techniques.

Although *PROJECTIVE* is the most efficient switch, it increases the ROM usage. Figures 3(a) and 3(b) show that the when the *PROJECTIVE* switch is enabled in case A, the ROM size is increased by 1,830, 1,822, and 2,236 bytes for MICAz, TelosB/Tmote Sky, and Imote2, respectively, while the RAM size doesn't change at all. In case B, as Figures 3(c) and 3(d) show, disabling the *PROJECTIVE* switch can save 2,396, 3,880, and 2,652 bytes in ROM for MICAz, TelosB/Tmote Sky, and Imote2, respectively. The *PROJECTIVE* switch is the most effective switch to speed up ECDSA operations, but it also incurs larger ROM consumption than any other optimization technique.

SHAMIR_TRICK is also an efficient option to speed up ECDSA signature verification. From Figure 2(e), we can see that the verification can be speed up by 2 times on all platforms when enabling *SHAMIR_TRICK* in case A. Both the ROM size and RAM size are increased. In case A, the RAM size is increased 634, 676, and 784 bytes for MICAz, TelosB/Tmote Sky, and Imote2, respectively. Similarly, the ROM size of MICAz, TelosB/Tmote Sky and Imote2 is increased 638, 632, and 620 bytes, respectively. In case B, disabling *SHAMIR_TRICK* makes verification 1.6 times slower but save 998, 2,068, and 876 bytes in ROM for MICAz, TelosB/Tmote Sky, and Imote2, respectively. The RAM size does not decrease much because the sliding window method is used for verification when *SHAMIR_TRICK* is disabled.

Now let us take a look at the *SLIDING_WIN* option. In case A, as Figures 2(c), 2(e), 3(a) and 3(b) show, enabling *SLIDING_WIN* can improve signature generation and verification 1.2 times faster at the cost of dramatic RAM increase (1,262, 1,328 and 1,472 bytes for MICAz, TelosB/Tmote Sky, and Imote2, respectively). In case B, as Figures 2(d), 2(f), 3(c) and 3(d) show, disabling *SLIDING_WIN* can save 632, 668 and 752 bytes of RAM usage for MICAz, TelosB/Tmote Sky, and Imote2 with 1.2 times slower signature generation and verification. Since MICAz and TelosB/Tmote Sky are low-end sensor platforms, they have much smaller RAM (4kB, 10kB) compared with Imote2 (256kB). Before enableing *SLIDING_WIN*, we should be very careful if the application requires large RAM consumption. Since *SLIDING_WIN* is the most RAM consuming switch in TinyECC, application developers may disable it or reduce the window size to reserve more RAM for the applications.



(a) Init. time when all other optimizations are disabled (case A)

(b) Init time when all other optimizations are enabled (case B)

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

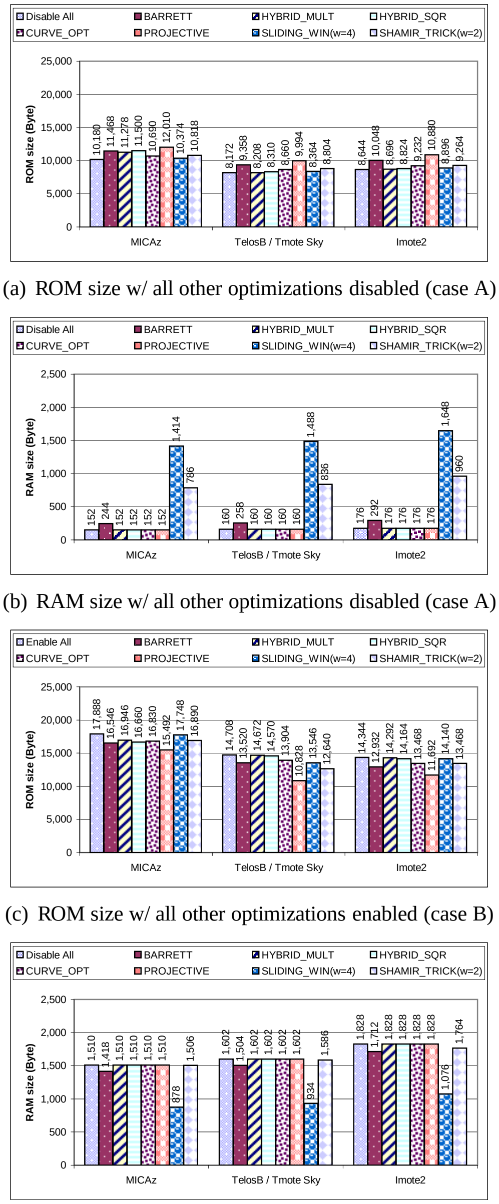
(d) Sig. generation time when all other optimizations are enabled (case B)

(e) Sig. verification time when all other optimizations are disabled (case A)

(f) Sig. verification time when all other optimizations are enabled (case B)

Figure 2. ECDSA timing result

Now consider the *HYBRID_MULT*, *HYBRID_SQR*, and *CURVE_OPT* options. In case A, *HYBRID_MULT*, *HYBRID_SQR* and *CURVE_OPT* do not have big impact on the timing result. However, in case B, *HYBRID_MULT* can speed up signature generation by 1.6 times for MICAz, 1.2 times for TelosB/Tmote Sky, and 1.2 times faster for Imote2. Similarly, it can speed up signature verification by 1.7 times for MICAz, 1.2 times for TelosB/Tmote Sky, and 1.2 times for Imote2. *HYBRID_SQR* can speed up signature generation by 1.5 times for MICAz, 1.2 times for TelosB/Tmote Sky, and 1.2 times for Imote2, and speed up signature verification by 1.5 times for MICAz, 1.2 times for TelosB/Tmote Sky, and 1.2 times faster for Imote2. *CURVE_OPT* can speed up signature generation by 2 times for MICAz, 1.9 times for TelosB/Tmote Sky, and 1.7 times for Imote2. Similarly, it can speed up signature verification by 2.1 times for MICAz, 1.9 times for TelosB/Tmote Sky, and 1.7 times for Imote2. The reason that *HYBRID_MULT*, *HYBRID_SQR* and *CURVE_OPT* cannot speed up ECDSA a lot in case A is that the *PROJECTIVE* option is disabled when each of these switches is enabled. Thus, inverse operation is the major computation of signature generation and verification. In case B, when *PROJECTIVE* is enabled, multiplication and squaring become the major computation in ECDSA.



(a) ROM size w/ all other optimizations disabled (case A)

(b) RAM size w/ all other optimizations disabled (case A)

(c) ROM size w/ all other optimizations enabled (case B)

(d) RAM size w/ all other optimizations enabled (case B)

Figure 3. Code size of ECDSA

Based on the timing results obtained for ECDSA, the effectiveness of these optimization switches in terms of execution time can be ordered as follows: *PROJECTIVE* $>$ *CURVE_OPT* $>$ *HYBRID_MULT* $>$ *HYBRID_SQR* $>$ *SLIDING_WIN* $>$ *SHAMIR_TRICK* $>$ *BARRETT*. In terms of RAM size, the optimization switches can be ordered as follows: *SLIDING_WIN* $>$ *SHAMIR_TRICK* $>$ *BARRETT* $>$ *HYBRID_MULT* $=$ *HYBRID_SQR* $=$ *CURVE_OPT* $=$ *PROJECTIVE*.

In terms of ROM size, the optimization switches are ordered differently for different platforms. For MICAz, *PROJECTIVE* $>$ *BARRETT* $\approx$ *HYBRID_SQR* $>$ *CURVE_OPT* $\approx$ *SHAMIR_TRICK* $\approx$ *HYBRID_MULT* $>$ *SLIDING_WIN*. For TelosB/Tmote Sky, *PROJECTIVE* $>$ *BARRETT* $\approx$ *SHAMIR_TRICK* $>$ *CURVE_OPT* $\approx$ *SLIDING_WIN* $>$ *HYBRID_SQR* $>$ *HYBRID_MULT*. For Imote2, *PROJECTIVE* $>$ *BARRETT* $>$ *SHAMIR_TRICK* $\geq$ *CURVE_OPT* $>$ *SLIDING_WIN* $>$ *HYBRID_SQR* $>$ *HYBRID_MULT*.

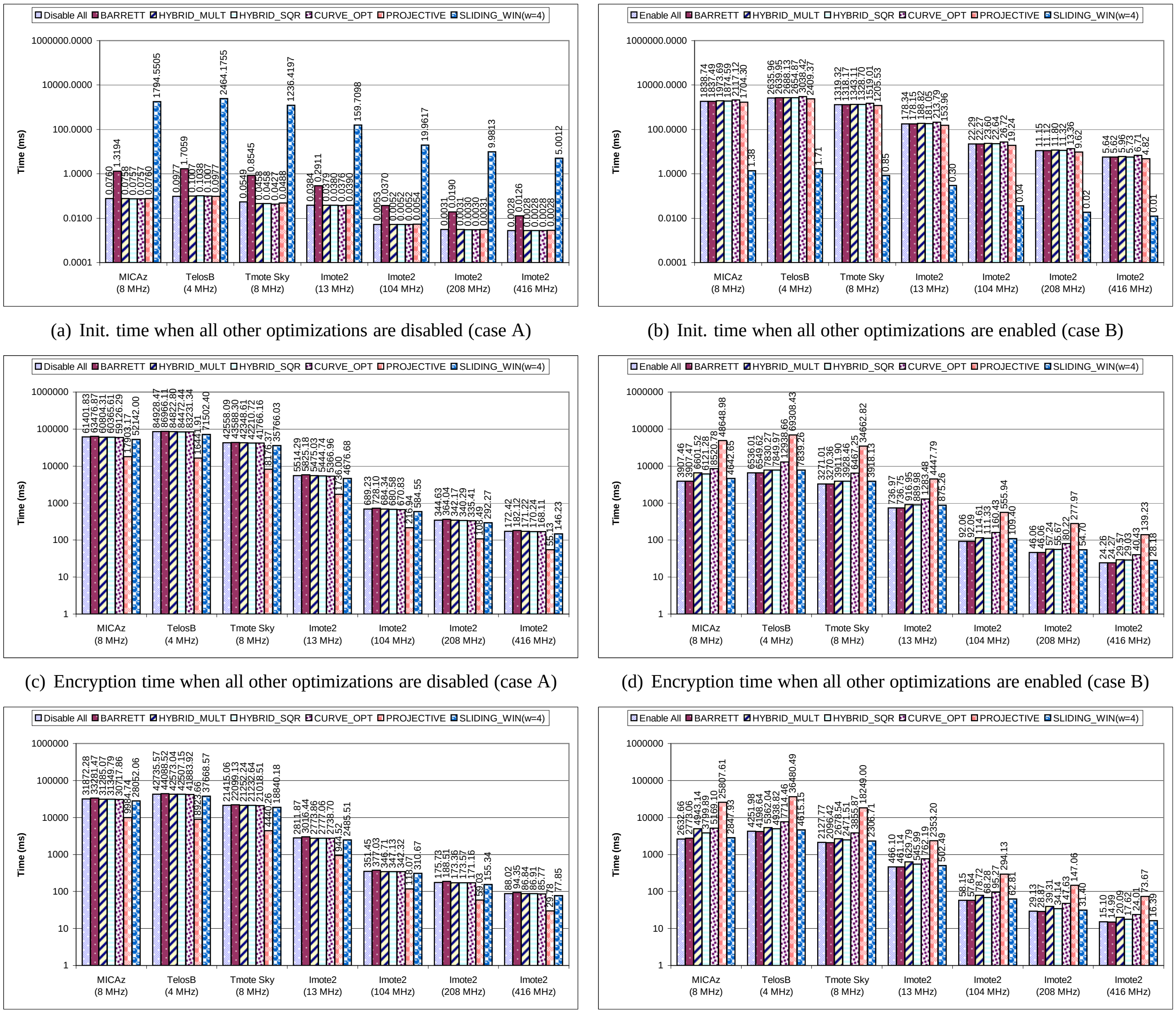
ECIES: Let us switch our attention to the performance results of ECIES. Figures 4 and 5 show the execution time and the storage requirements of ECIES, respectively. Since ECIES and ECDSA share the same implementation for basic elliptic curve operations, the effects of optimization switches are similar. Note that *SHAMIR_TRICK* is not applicable here.

In ECIES, only Barrett reduction and the sliding window method require precomputation. As Figure 4(a) shows, the precomputation of the sliding window method with window size 4 costs 1,795, 2,464, 1,236, and 160 ms for MICAz, TelosB, Tmote Sky, and Imote2 (13 MHz), respectively. In contrast, the precomputation for Barrett reduction only requires 1.3, 1.7, 0.9 and 0.3 ms for MICAz, TelosB, Tmote Sky, and Imote2 (13 MHz), respectively. This is because the sliding window method with window size 4 needs to precompute 16 points on the elliptic curve, but Barrett reduction only precomputes one large number μ.

The *PROJECTIVE* option is the most effective switch to speed up ECIES. As Figures 4(c) and 4(e) show, in case A, the *PROJECTIVE* option can speed up encryption 3.4, 5.2 and 3.2 times, and decryption 3.2, 4.8 and 3.0 times for MICAz, TelosB/Tmote Sky, and Imote2, respectively. Figures 4(d) and 4(f) show that, in case B, *PROJECTIVE* can speed up encryption 12.5, 10.6 and 6.0 times, can speed up decryption 9.8, 8.6 and 5.1 times for MICAz, TelosB/Tmote Sky, and Imote2, respectively.

The *PROJECTIVE* option is also the most ROM consuming switch in ECIES. In case A, as Figure 5(a) shows, the *PROJECTIVE* option increases ROM usage 1,032, 1,688 and 1,620 bytes for MICAz, TelosB/Tmote Sky, and Imote2, respectively. In case B, as Figure 5(c) shows, the *PROJECTIVE* option increases ROM usage 4,198, 4,310 and 4,988 bytes for MICAz, TelosB/Tmote Sky, and Imote2, respectively. In case A, as Figure 5(b) shows, the *PROJECTIVE* option does not require additional RAM usage. In case B, as Figure 5(d) shows, the *PROJECTIVE* option increase RAM usage 315, 330 and 360 bytes for MICAz, TelosB/Tmote Sky, and Imote2, respectively. Because sliding window method in projective coordinate system requires additional RAM for z axis.

The *SLIDING_WIN* option can speed up ECIES encryption by 1.2 times and speed up decryption by 1.1 times for all platforms in both cases. *SLIDING_WIN* is also the most RAM consuming switch as figures 5(b) and 5(d) show. For case A, *SLIDING_WIN* with window size 4 increases RAM usage 1,262, 1,328, 1,472 bytes for MICAz, TelosB/Tmote Sky, and



(a) Init. time when all other optimizations are disabled (case A)

(b) Init. time when all other optimizations are enabled (case B)

(c) Encryption time when all other optimizations are disabled (case A)

(d) Encryption time when all other optimizations are enabled (case B)

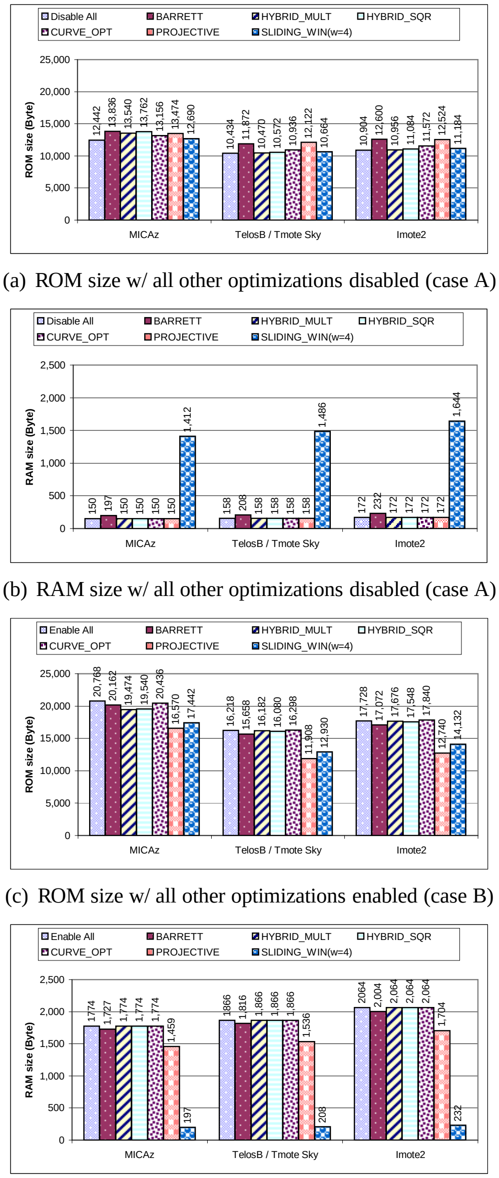
(e) Decryption time when all other optimizations are disabled (case A)

(f) Decryption time when all other optimizations are enabled (case B)

Figure 4. ECIES timing result

Imote2, respectively. For case B, *SLIDING_WIN* with window size 4 requires 1,577, 1,658, 1,832 bytes for MICAz, TelosB/Tmote Sky, and Imote2, respectively.

Figures 4(c) and 4(e) show that in case A, none of the remaining optimization techniques help much except *PROJECTIVE* option. But in case B, there are big differences, as reflected in Figures 4(d) and 4(f). In case B, when the *PROJECTIVE* option is enabled, the number of inverse operations is decreased a lot with increasing number of multiplications and squarings. Thus, the other optimization techniques can speed up ECIES in case B better than in case A. In case B, the *HYBRID_MULT* option can speed up encryption by 1.7 times for MICAz, 1.2 times for TelosB/Tmote Sky, and 1.4 times for Imote2. It can speed up decryption by 1.9 times for MICAz, 1.3 times for TelosB/Tmote Sky, and 1.6 times for Imote2. Similarly, the *HYBRID_SQR* option can speed up encryption by 1.5 times for MICAz, 1.2 times for TelosB/Tmote Sky, and 1.3 times for Imote2. It can speed up decryption by 1.4 times for MICAz, 1.2 times for TelosB/Tmote Sky, and 1.2 times for Imote2. The *CURVE_OPT* option can speed up encryption by 2.1 times for MICAz, 2.0 times for TelosB/Tmote Sky, and 1.7 times for Imote2. Similarly, it can speed up decryption by 2.0 times for MICAz, 1.8 times for TelosB/Tmote Sky, and 1.6 times for Imote2.



(a) ROM size w/ all other optimizations disabled (case A)

(b) RAM size w/ all other optimizations disabled (case A)

(c) ROM size w/ all other optimizations enabled (case B)

(d) RAM size w/ all other optimizations enabled (case B)

Figure 5. ECIES code size

Based on the above performance results, we can give an order of priority of the optimization switches when execution time is the primary objective as follows: *PROJECTIVE* > *CURVE_OPT* > *HYBRID_MULT* > *HYBRID_SQR* > *SLIDING_WIN* > *BARRETT*. In addition, the order of priority for the purpose of RAM size reduction is given as follows: *SLIDING_WIN* > *PROJECTIVE* > *BARRETT* > *HYBRID_MULT* = *HYBRID_SQR* = *CURVE_OPT*.

For ROM size, the optimization switches have different orders for different cases. For case A, TelosB/Tmote Sky and Imote2 have similar order: *PROJECTIVE* ≈ *BARRETT* > *CURVE_OPT* > *SLIDING_WIN* > *HYBRID_SQR* > *HYBRID_MULT*. MICAz has the order: *BARRETT* > *HYBRID_SQR* > *HYBRID_MULT* > *PROJECTIVE* > *CURVE_OPT* > *SLIDING_WIN*. For case B, TelosB/Tmote Sky have same order: *PROJECTIVE* > *SLIDING_WIN* > *BARRETT* > *HYBRID_SQR* > *HYBRID_MULT* > *CURVE_OPT*. MICAz has order: *PROJECTIVE* > *SLIDING_WIN* > *HYBRID_MULT* > *HYBRID_SQR* > *BARRETT* > *CURVE_OPT*.

ECDH: Figures 6 and 7 show the timing result and the storage requirements of ECDH, respectively. Similar to ECIES, the *SHAMIR_TRICK* option is not applicable to ECDH, either.

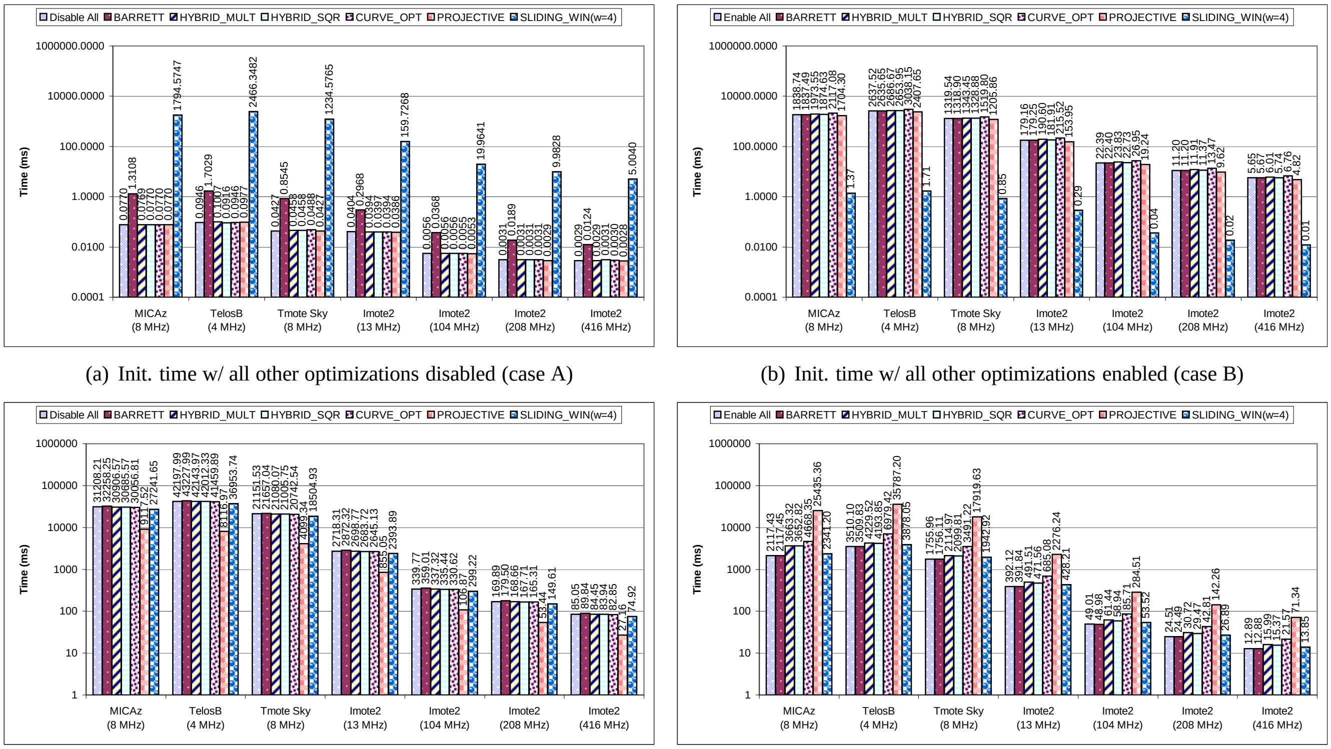
The *PROJECTIVE* option is the most efficient one for ECDH, though it is also the most ROM consuming among all the optimization switches. In case A, it can speed up ECDH key establishment 3.4, 5.2, and 3.2 times with additional 984, 1,144 ,and 1,552 bytes ROM requirement for MICAz, TelosB/Tmote Sky, and Imote2, respectively. In case B, it can speed up key establishment 12.0, 10.2, and 5.8 times with 3,920, 4,220, and 4,864 bytes more ROM usage for MICAz, TelosB/Tmote Sky and Imote2, respectively.

The *HYBRID_MULT*, *HYBRID_SQR* and *CURVE_OPT* options do not work well in case A due to the disabling of the *PROJECTIVE* option. In case B, the *HYBRID_MULT* option can speed up key agreement 1.7, 1.2, and 1.3 times with 1,345, 36, and 52 more bytes ROM usage for MICAz, TelosB/Tmote Sky, and Imote2, respectively. The *HYBRID_SQR* option can speed up key agreement 1.7, 1.2, and 1.2 times with 1,228, 138, and 180 more bytes ROM usage for MICAz, TelosB/Tmote Sky, and Imote2, respectively. Finally, the *CURVE_OPT* option can speed up key agreement 2.2, 2.0, and 1.8 times with 86 more, but 204 and 264 fewer bytes ROM usage for MICAz, TelosB/Tmote Sky, and Imote2, respectively. This is reasonable because nesC compiler may do different optimizations for different platforms. We also use more inline assembly code in specific curve optimization to reduce the memory operation, but do not have such inline assembly code specifically for *CURVE_OPT* on TelosB/Tmote Sky and Imote2.

The *SLIDING_WIN* option can speed up key establishment by 1.1 times faster in both cases. *SLIDING_WIN* is the most RAM consuming switch as figures 7(b) and 7(d) show. For case A, *SLIDING_WIN* increases the RAM usage 1,262, 1,328, and 1,472 bytes for MICAz, TelosB/Tmote Sky, and Imote2, respectively. For case B, *SLIDING_WIN* increases the RAM usage 1,577, 1,658, and 1,832 bytes for MICAz, TelosB/Tmote Sky, and Imote2, respectively.

According to the effectiveness of speeding up ECDH, we should enable each optimization switches in the following order: *PROJECTIVE* > *CURVE_OPT* > *HYBRID_MULT* > *HYBRID_SQR* > *SLIDING_WIN* > *BARRETT*.

For required RAM size, the optimization switches has the following order for all platforms: *SLIDING_WIN* > *PROJEC-*



(a) Init. time w/ all other optimizations disabled (case A)

(b) Init. time w/ all other optimizations enabled (case B)

(c) Key establishment time w/ all other optimizations disabled (case A)

(d) Key establishment time w/ all other optimizations enabled (case B)

Figure 6. ECDH timing result

TIVE $>$ *BARRETT* $>$ *HYBRID_MULT* $=$ *HYBRID_SQR* $=$ *CURVE_OPT*.

For required ROM size, the optimization switches has different order for different platforms in different cases. TelosB/Tmote Sky and Imote2 have same orders in both cases. In case A, they have order: *PROJECTIVE* $>$ *BARRETT* $>$ *SLIDING_WIN* $>$ *HYBRID_SQR* $>$ *HYBRID_MULT* $>$ *CURVE_OPT*. In case B, they have order: *PROJECTIVE* $>$ *SLIDING_WIN* $>$ *BARRETT* $>$ *HYBRID_SQR* $>$ *HYBRID_MULT* $>$ *CURVE_OPT*. MICAz has different orders. In case A, it has order: *HYBRID_SQR* $>$ *HYBRID_MULT* $>$ *BARRETT* $>$ *PROJECTIVE* $>$ *SLIDING_WIN* $>$ *CURVE_OPT*. In case B, it has order: *PROJECTIVE* $>$ *SLIDING_WIN* $>$ *HYBRID_MULT* $>$ *HYBRID_SQR* $>$ *BARRETT* $>$ *CURVE_OPT*.

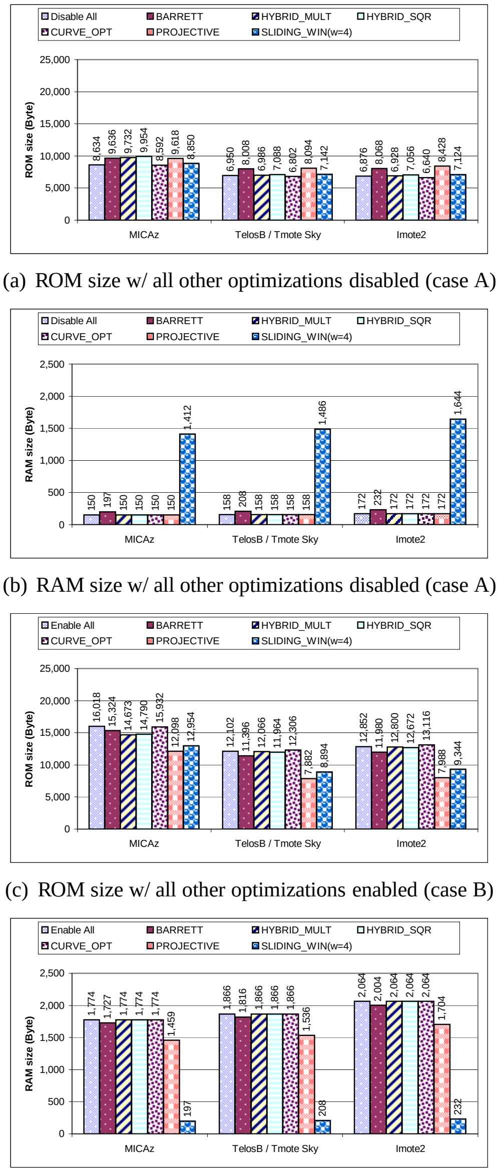
6.2.2. Most Computationally Efficient Configuration

Now let us take a closer look at the most computationally efficient configuration. Apparently, TinyECC provides the most computationally efficient configuration when all the optimization switches are enabled. Figure 8 shows the execution time required by ECDSA initialization, signature generation, signature verification; ECIES initialization, encryption, decryption; ECDH initialization, key establishment.

From figure 8, we can see that enabling all optimization switches requires long pre-computation. For example, it takes MICAz 3,493, 1,839 and 1,839 ms to do pre-computation for ECDSA, ECIES and ECDH, respectively. Most of the pre-computation time is for the sliding window method and Shamir's trick (ECDSA only). It even takes longer time for TelosB to do pre-computation because TelosB can only run at 4 MHz. TelosB is slower than MICAz and Imote2 in ECDSA, ECIES, and ECDH operations. Tmote Sky, which runs at 8 MHz, is two times faster than TelosB. Running at 13 MHz, the default CPU frequency for Imote2, Imote2 is faster than MICAz in all operations. If we set the frequency to 416 MHz, it only takes 12 and 14 ms to generate ECDSA signature and verify it. Moreover, it can perform ECIES encryption in 24 ms and decrypt in 15 ms. Finally, ECDH key establishment only takes 13 ms.

Enabling all optimization switches requires the largest ROM and RAM consumptions. Figure 9 shows the ROM and RAM requirements by all schemes. Imote2 has the largest RAM size due to its word size. MICAz has the smallest RAM size due to its 8-bit word size, but it has the largest ROM size because it has additional assembly code for minimizing memory operation when *CURVE_OPT* option is enabled.

Now consider the energy consumption of ECDSA, ECIES



(a) ROM size w/ all other optimizations disabled (case A)

(b) RAM size w/ all other optimizations disabled (case A)

(c) ROM size w/ all other optimizations enabled (case B)

(d) RAM size w/ all other optimizations enabled (case B)

Figure 7. ECDH code size

and ECDH on the testing platforms. We compute energy consumption using $W = U \times I \times t$, where U is the voltage, I is the current draw in active mode with radio off, and t is the execution time. We took the voltage and current draw (with radio off) from the data sheet of each sensor platform [1,2,5,7], and used the execution time obtained in our experiments. Specifically, we chose U as 3v for MICAz, TelosB and Tmote Sky. The current draw for MICAz and TelosB/Tmote Sky was 8 mA and 1.8 mA, respectively. For Imote2, U is 0.95v for 13 MHz and 104 MHz [1]. The Imote2 data sheet [1] does not give the current draw when the node runs at 104 MHz with radio off. To be conservative, we use the current draw with radio on in our computation. That is, we chose 31 mA and 66 mA for Imote2 at 13 MHz and 104 MHz.

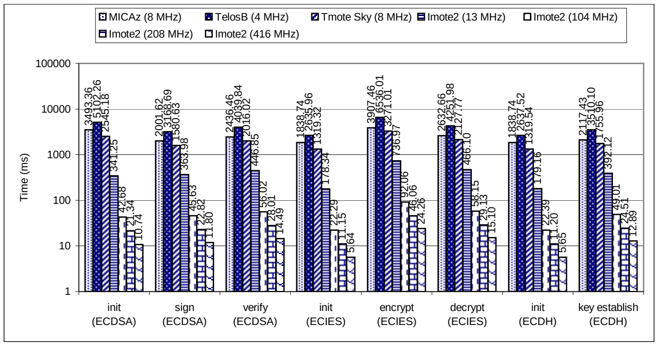
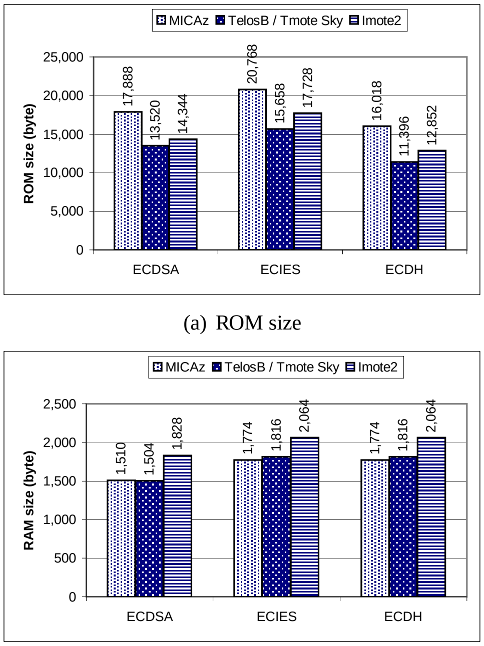


Figure 8. Execution time of ECDSA, ECIES, and ECDH w/ all optimization switches enabled



(a) ROM size

(b) RAM size

Figure 9. Code size of ECDSA, ECIES, and ECDH w/ all optimization switches enabled

Figure 10 shows the energy consumption required by all these operations. Imote2 is the most energy efficient platform when it runs at 104 MHz. It needs 2.86 mJ and 3.51 mJ to generate ECDSA signature and verify it; it needs 5.77 mJ and 3.65 mJ to do ECIES encryption and decryption; and it needs 3.07 mJ for the ECDH key agreement operation. MICAz is the most energy consuming platform. TelosB is quite efficient at energy consumption due to its low current draw with radio off. Tmote Sky consumes half as TelosB does because Tmote

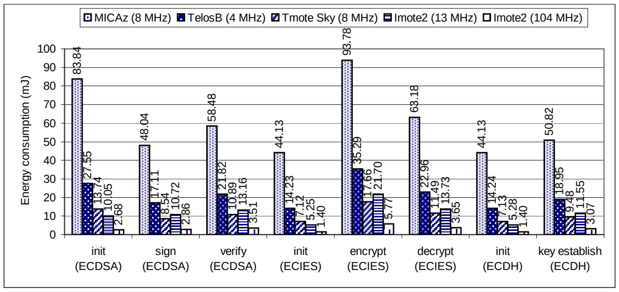


Figure 10. Energy consumption of ECDSA, ECIES, and ECDH w/ all optimization switches enabled

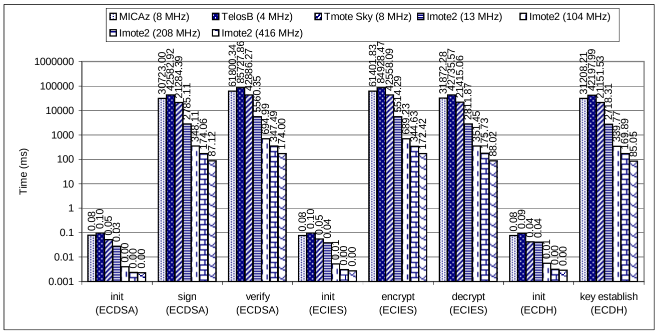


Figure 11. Execution time of ECDSA, ECIES, and ECDH w/ all optimization switches disabled

Sky (8 MHz) is two times faster than TelosB (4 MHz).

6.2.3. Most Storage-Efficient Configuration

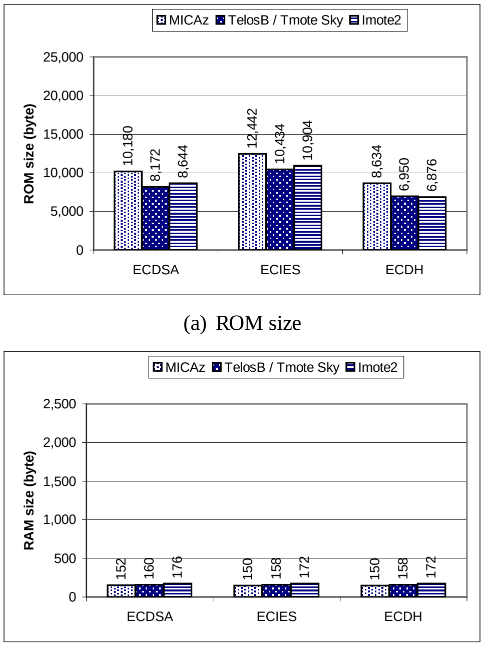
Many TinyOS applications may use TinyECC for authentication, encryption/decryption, or key establishment. Thus it is likely that TinyECC will be loaded on sensor nodes with other applications. Due to the resource constraint of low-end sensor platforms (e.g., MICAz, TelosB/Tmote Sky), we may have to reduce ROM and RAM size by disabling some optimization techniques to reserve enough space for other TinyOS applications.

For example, when all optimization switches are enabled, ECDSA needs 17,888 bytes ROM and 1,510 bytes RAM on MICAz, as figure 9 shows. Stack overflow may happen when TinyECC is integrated with other programs such as TOSBase; the available stack for local variables may not be large enough due to the limited RAM (4K bytes) on MICAz. As another example, TelosB only has 48K bytes ROM. If ECDSA with all optimizations enabled is integrated with the SurgeTelos, the total ROM size would be 40,380 bytes, leaving little space for other applications.

We can disable all optimization switches to show how compact TinyECC could be. Figure 11 shows the execution time of ECDSA, ECIES and ECDH when all optimization switches are disabled. In this case, no pre-computation is needed, and the initialization time is close to 0. Imote2 running at 416 MHz is still the fastest one. It can perform ECDSA signature generation and verification in 87 ms and 174 ms, respectively, and perform ECIES encryption and decryption in 172 ms and 88 ms, respectively, and finish ECDH key agreement in 85 ms. TelosB is the slowest platform due to the 4 MHz running frequency. It needs 42,583 ms and 85,728 ms for ECDSA signature generation and verification, respectively. ECIES encryption and decryption require 84,928 ms and 42,736 ms, respectively. ECDH key establishment can be done in 42,198 ms. Tmote Sky is two times faster than TelosB. MICAz is faster than TelosB but slower than Tmote Sky.

The benefit of disabling all optimization switches is the compact code size. Figure 12 shows the code size of all schemes in TinyECC when all optimization switches are disabled. Due to their word size, Imote2 has the largest RAM size, while MICAz has the smallest RAM size. The code size has been reduced greatly. For MICAz, the ROM size has been reduced by 7,708, 8,326, and 7,384 bytes for ECDSA, ECIES, and ECDH, respectively; the RAM size has been reduced by 1,358, 1,624, and 1,624 bytes for ECDSA, ECIES, and ECDH, respectively. For TelosB, the ROM size has been reduced by 5,348, 5,224, and 4,446 bytes for ECDSA, ECIES, and ECDH, respectively. Similarly, the RAM size has been reduced by 1,344, 1,658, and 1,658 bytes for ECDSA, ECIES, and ECDH, respectively. The developer can further reduce ROM size of ECDH by enabling *CURVE_OPT* as figures 7(a) and 7(c) show, but this does not work for ECDSA and ECIES.

Since the execution time of TinyECC is much longer, the energy consumption of TinyECC is also increased as figure 13 shows. Even when Imote2 runs at 104 MHz, it needs 21.83 mJ to generate ECDSA signature, which is almost 7.6 times more than the most computation-efficient case. For MICAz, it requires almost 15.4 times more energy to generate an ECDSA signature and 25.4 times more energy to verify a signature than it does in the most computation efficient case. Moreover, a node needs 15.7 times more energy for ECIES encryption, 12.1 times more energy for decryption, and 14.7 times more energy to establish a key in ECDH. For TelosB/Tmote Sky, a node needs 13.4 and 21.2 times more energy for ECDSA signature generation and verification, 13.0 and 10.1 times more energy for ECIES encryption and decryption, and 12.0 times more energy for ECDH key establishment. For Imote2, it needs 7.6 and 12.4 times more energy for ECDSA signature generation and verification, 7.5 and 6.0 times more energy for ECIES encryption and decryption, and 6.9 times more energy for ECDH key establishment.



(a) ROM size

(b) RAM size

Figure 12. Code size of ECDSA, ECIES, and ECDH w/ all optimization switches disabled

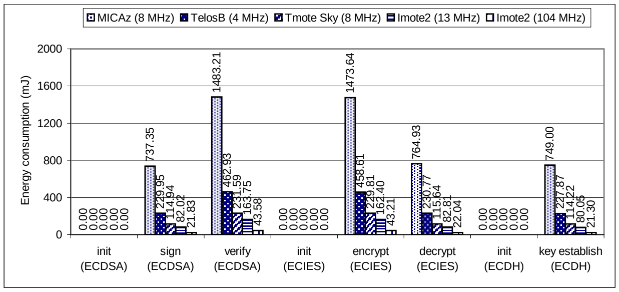


Figure 13. Energy consumption of ECDSA, ECIES, and ECDH w/ all optimization switches disabled

7. Related Work

A comprehensive guide for elliptic curve cryptography is given in [20]. A brief introduction to ECC can be found in [41]. Additional documentation on ECC can be found in [9–11]. There have been numerous ECC implementations in various contexts (e.g., Crypto++ [13], OpenSSL [3], MIRACL [39], NSS [34]). Most of these implementations are aimed at traditional computing platforms such as PCs.

Several recent efforts have focused on sensor platforms, such as the Mica series of motes. Malan et al. implemented ECC over binary extension fields F_{2^m} on TinyOS for Mica2 [31]. Unfortunately, due to the constraints on the typical microprocessors used by sensors, it is difficult to obtain efficient ECC over F_{2^m}. Gura et al. implemented and compared ECC and RSA on Atmel ATmega128 in assembly [19]. However, it is not clear how well their implementation can be integrated into sensor applications. Wang et al. implemented ECC on specific 160-bit elliptic curves on MICAz and TelosB running TinyOS [40]. They were able to obtain very fast execution time by hard-coding all the curve parameters into assembly code.

A common limitation of all these efforts is that all these attempts were developed as independent packages/applications without seriously considering the resource demands of sensor network applications. As a result, developers may found it difficult, and sometimes impossible, to integrate an ECC implementation with the sensor network applications (e.g., not enough ROM or RAM), though the ECC implementation may be okay on its own. In contrast, TinyECC provides a set of optimization switches that allow itself to be configured with different resource consumptions. This allows TinyECC to be flexibly integrated into sensor network applications.

8. Conclusion

In this paper, we presented the design, implementation, and evaluation of TinyECC, a *configurable* library for ECC operations in wireless sensor networks. A unique feature of TinyECC is its *configurability*. It provides a number of optimization switches, which can turn specific optimizations on or off based on developers' needs. Different combinations of the optimizations have different execution time and resource consumptions, and thus give the developers great flexibility in integrating TinyECC into sensor network applications. We also performed a series of experiments to evaluate the performance and resource consumptions of TinyECC with different combinations of enabled optimizations. In particular, our experimental results gave the most computationally efficient and the most storage efficient configurations of TinyECC.

In our future work, we plan to investigate techniques that can further speed up the execution and reduce the resource consumption for ECC-based PKC operations. We will also explore opportunities that can harness resources on high-end sensors in hybrid sensor networks.

References

[1] Imote2: High-performance wireless sensor network node. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Imote2_Datasheet.pdf.

[2] MICAz: Wireless measurement system. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[3] The openssl project. http://www.openssl.org/.

[4] SSL 3.0 specification. http://wp.netscape.com/eng/ssl3/.

[5] TelosB mote platform. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf.

[6] TinyOS: An open-source OS for the networked sensor regime. http://www.tinyos.net/.

[7] Tmote sky: Reliable low-power wireless sensor networking eases development and deployment. http://www.moteiv.com/products-tmotesky.php.

[8] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks*, 38(4):393–422, 2002.

[9] American Bankers Association. *ANSI X9.62-1998: Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)*, 1999.

[10] Certicom Research. Standards for efficient cryptography – SEC 1: Elliptic curve cryptography. http://www.secg.org/download/aid-385/sec1_final.pdf, September 2000.

[11] Certicom Research. Standards for efficient cryptography – SEC 2: Recommended elliptic curve domain parameters. http://www.secg.org/collateral/sec2_final.pdf, September 2000.

[12] H. Chan, A. Perrig, and D. Song. Random key predistribution schemes for sensor networks. In *IEEE Symposium on Research in Security and Privacy*, pages 197–213, 2003.

[13] W. Dai. Crypto++ library 5.5. http://www.cryptopp.com/, May 2007.

[14] J. Deng, R. Han, and S. Mishra. Secure code distribution in dynamically programmable wireless sensor networks. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks (IPSN '06)*, April 2006.

[15] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, November 1976.

[16] W. Du, J. Deng, Y. S. Han, and P. Varshney. A pairwise key pre-distribution scheme for wireless sensor networks. In *Proceedings of 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 42–51, October 2003.

[17] L. Eschenauer and V. D. Gligor. A key-management scheme for distributed sensor networks. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 41–47, November 2002.

[18] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI '03)*, June 2003.

[19] N. Gura, A. Patel, and A. Wander. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Proceedings of the 2004 Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004)*, pages 119–132, August 2004.

[20] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[21] S. Kent and R. Atkinson. IP authentication header. IETF RFC 2402, November 1998.

[22] S. Kent and R. Atkinson. IP encapsulating security payload (ESP). IETF RFC 2406, November 1998.

[23] D.E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, third edition, 1997. ISBN: 0-201-89684-2.

[24] RSA Laboratories. RSAREF: A cryptographic toolkit (version 2.0), March 1994.

[25] P.E. Lanigan, R. Gandhi, and P. Narasimhan. Sluice: Secure dissemination of code updates in sensor networks. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06)*, July 2006.

[26] A. Liu, P. Kampanakis, and P. Ning. TinyECC: Elliptic curve cryptography for sensor networks (version 0.3). http://discovery.csc.ncsu.edu/software/TinyECC/.

[27] D. Liu and P. Ning. Establishing pairwise keys in distributed sensor networks. In *Proceedings of 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 52–61, October 2003.

[28] D. Liu and P. Ning. Multi-level μTESLA: Broadcast authentication for distributed sensor networks. *ACM Transactions in Embedded Computing Systems (TECS)*, 3(4):800–836, 2004.

[29] D. Liu and P. Ning. Improving key pre-distribution with deployment knowledge in static sensor networks. *ACM Transactions on Sensor Networks*, 1(2):204–239, November 2005.

[30] D. Liu, P. Ning, S. Zhu, and S. Jajodia. Practical broadcast authentication in sensor networks. In *Proceedings of the 2nd Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous 2005)*, July 2005.

[31] D. Malan, M. Welsh, and M. Smith. A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography. In *Proceedings of IEEE Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, pages 71–80, 2004.

[32] K. Malasri and L. Wang. Addressing security in medical sensor networks. In *HealthNet '07: Proceedings of the 1st ACM SIGMOBILE international workshop on Systems and networking support for healthcare and assisted living environments*, pages 7–12, 2007.

[33] A.J. Menezes, P. C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN: 0-8493-8523-7.

[34] Mozilla. Network security service (NSS). http://www.mozilla.org/projects/security/pki/nss/.

[35] National Institute of Standards and Technology. Digital signature standard. Federal Information Processing Standard 186, http://csrc.nist.gov/publications/., 1993.

[36] National Institute of Standards and Technology. Recommended elliptic curves for federal government use, August 1999.

[37] A. Perrig, R. Canetti, D. Song, and D. Tygar. Efficient authentication and signing of multicast streams over lossy channels. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000.

[38] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and D. Tygar. SPINS: Security protocols for sensor networks. In *Proceedings of Seventh Annual International Conference on Mobile Computing and Networks*, pages 521–534, July 2001.

[39] Shamus Software. Multiprecision integer and rational arithmetic c/c++ library (MIRACL). http://www.shamus.ie/.

[40] H. Wang and Q. Li. Efficient implementation of public key

cryptosystems on mote sensors. In *Proceedings of International Conference on Information and Communication Security (ICICS)*, pages 519–528, Dec. 2006.

[41] Wikipedia. Elliptic curve cryptography. `http://en.wikipedia.org/wiki/Elliptic_curve_cryptography`. Visited on May 23rd, 2007.