A Public-Key Infrastructure for Key Distribution in TinyOS Based on Elliptic Curve Cryptography

David J. Malan, Matt Welsh, Michael D. Smith
Division of Engineering and Applied Sciences
Harvard University
{malan,mdw,smith}@eecs.harvard.edu

***Abstract*— We present the first known implementation of elliptic curve cryptography over $\mathbb{F}_{2^p}$ for sensor networks based on the 8-bit, 7.3828-MHz MICA2 mote. Through instrumentation of UC Berkeley's TinySec module, we argue that, although secret-key cryptography has been tractable in this domain for some time, there has remained a need for an efficient, secure mechanism for distribution of secret keys among nodes. Although public-key infrastructure has been thought impractical, we argue, through analysis of our own implementation for TinyOS of multiplication of points on elliptic curves, that public-key infrastructure is, in fact, viable for TinySec keys' distribution, even on the MICA2. We demonstrate that public keys can be generated within 34 seconds, and that shared secrets can be distributed among nodes in a sensor network within the same, using just over 1 kilobyte of SRAM and 34 kilobytes of ROM.**

I. Introduction

Wireless sensor networks have been proposed for such applications as habitat monitoring [1], structural health monitoring [2], emergency medical care [3], and vehicular tracking [4], all of which demand some combination of authentication, integrity, privacy, and security. Unfortunately, the state of the art has offered weak, if any, guarantees of these needs.

The limited resources boasted by today's sensor networks appear to render them ill-suited for the most straightforward implementations of security protocols. Consider the MICA2 mote [5], designed by researchers at the University of California at Berkeley and fabricated by Crossbow Technology, Inc. This device offers an 8-bit, 7.3828-MHz ATmega 128L processor, 4 kilobytes (KB) of primary memory (SRAM), and 128 KB of program space (ROM). Such a device, given these resources, is seemingly unfit for computationally expensive or energy-intensive operations. For this reason has public-key cryptography often been ruled out for sensor networks as an infrastructure for authentication, integrity, privacy, and security [6]–[9], even despite its allowance for secure rekeying of mobile devices.

But such conclusions have been backed too infrequently by actual data. In fact, to our knowledge, little empirical research has been published on the viability of public-key infrastructure (PKI) for the MICA2, save for a cursory analysis of an implementation of RSA [10] and a recent comparison of RSA and elliptic curve cryptography (ECC) over $\mathbb{F}_p$ [11].

Our work aspires to fill this void. Through instrumentation of TinyOS, we first demonstrate that secret-key cryptography is tractable on the MICA2. By way of our own implementation of multiplication of points on elliptic curves, we then argue that PKI for secret keys' distribution is, in fact, tractable as well. Public keys can be generated within 34 seconds (sec), and shared secrets can be distributed within the same, using just over 1 KB of SRAM and 34 KB of ROM.

We begin these arguments in Section II with an analysis of TinySec [6], TinyOS's existing secret-key infrastructure for the MICA2 based on SKIPJACK [12]. In Section III, we address shortcomings in that infrastructure with a look at an implementation of Diffie-Hellman for the MICA2 based on the Discrete Logarithm Problem (DLP) and expose weaknesses in its design for sensor networks. In Section IV, we redress those weaknesses with our own implementation of Diffie-Hellman based on the Elliptic Curve Discrete Logarithm Problem (ECDLP). In Section V, we discuss optimizations underlying our implementation. In Section VI, we propose directions for future work, while, in Section VII, we explore related work. We conclude in Section VIII.

II. SKIPJACK and the MICA2

TinyOS currently offers the MICA2 access control, authentication, integrity, and confidentiality through TinySec, a link-layer security mechanism based on SKIPJACK in cipher-block chaining mode. An 80-bit symmetric cipher, SKIPJACK is the formerly classified algorithm behind the Clipper chip, approved by the National Institute for Standards and Technology (NIST) in 1994 for the Escrowed Encryption Standard [13]. TinySec supports message authentication and integrity with message authentication codes, confidentiality with encryption, and access control with shared, group keys.

The mechanism allows for an 80-bit key space, the benefit of which is that known attacks require as many 2^{79} operations on average (assuming SKIPJACK isn't reduced from 32

rounds [14]).[1] Moreover, as packets under TinySec include a 4-byte message authentication code (MAC), the probability of blind forgery is only 2^{-32}. This security comes at a cost of just five bytes (B): whereas transmission of some 29-byte plaintext and its cyclic redundancy check (CRC) requires a packet of 36 B, transmission of that plaintext's ciphertext and MAC under TinySec requires a packet of only 41 B, as the mechanism borrows TinyOS's fields for Group ID (TinyOS's weak, default mechanism for access control) and CRC for its MAC.

Performance. The impact of TinySec on the MICA2's performance is reasonable. On first glance, it would appear that TinySec adds under 2 milliseconds (ms) to a packet's transmission time (Table I) and under 5 ms to a packet's round-trip time to and from some neighbor (Table II). However, the apparent overhead of TinySec, 1,244 microseconds (μsec) on average, as suggested by transmission times, is nearly subsumed by the data's root mean square (1,094 μsec). Round-trip times exhibit less variance, but tighter benchmarks are in order for TinySec's accurate analysis.

Table III, then, offers results with yet less variance from finer instrumentation of TinySec: encryption of a 29-byte, random payload requires 2,190 μsec on average, and computation of that payload's MAC requires 3,049 μsec on average; overall, TinySec adds 5,239 $\pm$ 18 μsec to a packet's computational requirements. It appears, then, that some of those cycles can be subsumed by delays in scheduling and medium access, at least for applications not already operating at full duty. Fig. 1, the results of an analysis of the MICA2's throughput, without and with TinySec enabled, puts the mechanism's computational overhead for such applications into perspective: on average, TinySec may lower throughput of acknowledged packets by only 0.28 packets per second. These results appear in line with UC Berkeley's own evaluation of TinySec [15].

Memory. Of course, TinySec's encryption and authentication does come at an additional cost in memory. Per Table IV, TinySec adds 454 B to an application's `.bss` segment, 276 B to an application's `.data` segment, 7,076 B to an application's `.text` segment, and 92 B to an application's maximal stack size during execution. For applications that don't require the entirety of the MICA2's 128 KB of program memory and 4 KB of primary memory, then, TinySec is a viable addition.

Security. As with any cipher based only on shared secrets, TinySec is, of course, vulnerable to various attacks. After all, the MICA2 is intended for deployment in sensor networks. For reasons of cost and logistics, long-term, physical security of the devices is unlikely. Compromise of the network, therefore, reduces to compromise of any one node, unless, for instance, rekeying is possible. Pairwise keys among n nodes would certainly provide some defense against compromises of individual nodes. But n^2 80-bit keys would more than exhaust a node's SRAM for n as small as 20. A more sparing use of secret keys is in order, but secure, dynamic establishment of those keys, particularly for networks in which the positions of sensors may be transient, requires a chain or infrastructure of trust. In fact, the very design of TinySec requires as much for rekeying as well. Though TinySec's 4-byte initialization vector (IV) allows for secure transmission of some message as many as 2^{32} times, that bound may be insufficient for embedded networks whose lifespans demand longer lasting security.[2] Needless to say, TinySec's reliance on a single secret key prohibits the mechanism from securely rekeying itself.

Fortunately, these problems of secret keys' distribution are redressed by public-key infrastructure. The sections that follow thus explore options for that infrastructure's design and implementation on the MICA2.

TABLE I

TRANSMISSION TIMES REQUIRED TO TRANSMIT A 29-BYTE, RANDOM PAYLOAD, AVERAGED OVER 1,000 TRIALS, WITH AND WITHOUT TINYSEC ENABLED. TRANSMISSION TIME IS DEFINED HERE AS THE TIME ELAPSED BETWEEN `SendMsg.send(·,·,·)` AND `SendMsg.sendDone()`. THE IMPLIED OVERHEAD OF TINYSEC ON TRANSMISSION TIME IS GIVEN AS THE DIFFERENCE OF THE DATA'S MEANS. THE ROOT MEAN SQUARE IS DEFINED AS $\sqrt{s_{\text{W/O}}^2/1,000 + s_{\text{W/}}^2/1,000}$, WHERE $s_{\text{W/O}}$ AND $s_{\text{W/}}$ ARE THE DATA'S STANDARD DEVIATIONS.

	without TinySec	with TinySec
Median	72,904 μsec	74,367 μsec
Mean	74,844 μsec	76,088 μsec
Standard Deviation	24,248 μsec	24,645 μsec
Standard Error	767 μsec	779 μsec

Implied Overhead of TinySec	1,244 μsec
Root Mean Square	1,094 μsec

III. DLP AND THE MICA2

With the utility of SKIPJACK-based TinySec thus motivated and the mechanism's costs exposed, we next examine DLP, on which Diffie-Hellman [16] is based, as an answer to the MICA2's problems of secret keys' distribution. DLP typically involves recovery of $x \in \mathbb{Z}_p$, given p, g, and $g^x \pmod p$, where p is a prime integer, and g is a generator of $\mathbb{Z}_p$. By leveraging the presumed difficultly of DLP, Diffie-Hellman allows two parties to agree, without prior arrangement, upon a shared secret, even in the midst of eavesdroppers, with perfect forward secrecy, as depicted in Fig. 2. Authenticated exchanges

[1] Although TinySec allows for 80-bit keys, its current implementation actually relies on 64-bit keys that are extended with 16 bits of padding.

[2] To allow for secure transmission of as many as 2^{32} packets, it is actually necessary to modify TinySec so that it no longer writes a mote's address into the third and fourth bytes of a mote's IV.

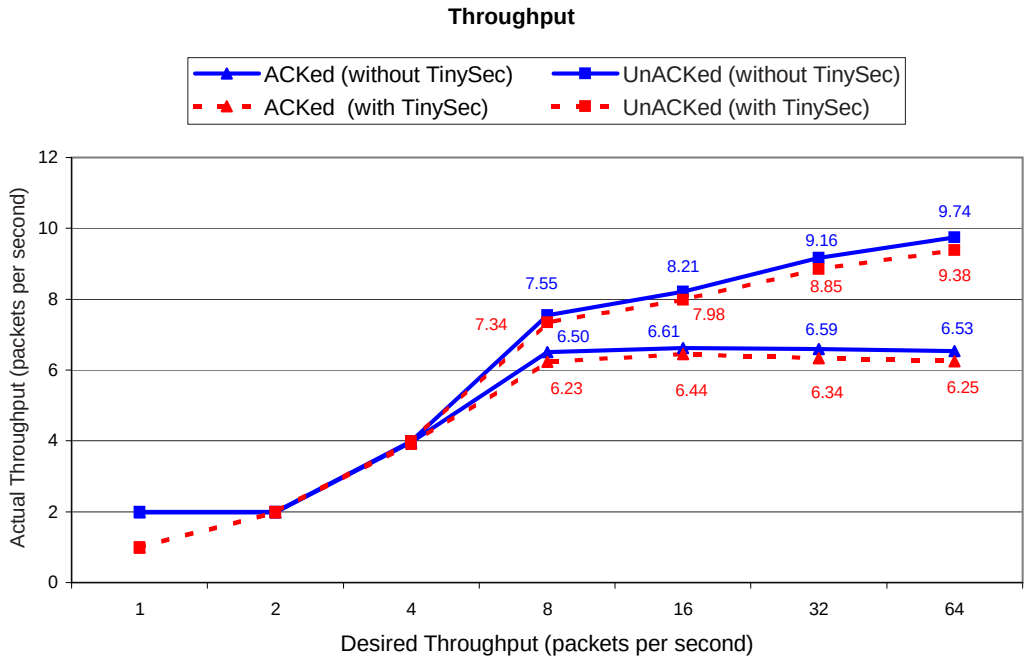

Fig. 1. Actual throughput versus desired throughput for acknowledged (ACKed) and unacknowledged (unACKed) transmissions between a sender and a receiver, averaged over ten minutes of transmission per level of desired throughput, where desired throughput is the rate at which calls to `SendMsg.send(·,·,·)` were scheduled by `Timer.start(·,·)`. ACKed actual throughput is the rate at which 29-byte, random payloads from a sender were received and subsequently acknowledged by an otherwise passive recipient. UnACKed actual throughput is the rate at which the sender actually sent such packets, acknowledged or not (*i.e.*, the rate at which calls to `SendMsg.send(·,·,·)` were actually processed). For clarity, where ACKed and unACKed throughput begins to diverge are points labelled with values for actual throughput. In environments with less contention for medium access than in ours, higher throughput is possible, with and without TinySec enabled.

TABLE II

ROUND-TRIP TIMES REQUIRED TO TRANSMIT A 29-BYTE, RANDOM PAYLOAD, WITH AND WITHOUT TINYSEC ENABLED, FROM ONE NODE TO A NEIGHBOR AND BACK AGAIN, AVERAGED OVER 1,000 TRIALS. MORE PRECISELY, ROUND-TRIP TIME IS DEFINED HERE AS THE TIME ELAPSED BETWEEN `SENDMSG.SEND(·,·,·)` AND `RECEIVEMSG.RECEIVE(·)`. THE IMPLIED OVERHEAD OF TINYSEC ON ROUND-TRIP TIME IS GIVEN AS THE DIFFERENCE OF THE DATA'S MEANS. THE ROOT MEAN SQUARE IS DEFINED AS $\sqrt{s_{\text{w/o}}^2/1,000 + s_{\text{w/}}^2/1,000}$, WHERE $s_{\text{w/o}}$ AND $s_{\text{w/}}$ ARE THE DATA'S STANDARD DEVIATIONS.

	without TinySec	with TinySec
Median	145,059 μsec	149,290 μsec
Mean	147,044 μsec	152,015 μsec
Standard Deviation	30,736 μsec	31,466 μsec
Standard Error	972 μsec	995 μsec

Implied Overhead of TinySec	4,971 μsec
Root Mean Square	1,391 μsec

are possible with the station-to-station protocol (STS) [17], a variant of Diffie-Hellman.

With a form of Diffie-Hellman, then, could two nodes thus establish a shared secret for use as TinySec's key. At issue, though, is the cost of such establishment on the MICA2.

TABLE III

TIMES REQUIRED TO TO ENCRYPT A 29-BYTE, RANDOM PAYLOAD, AND TO COMPUTE THAT PAYLOAD'S MAC, AVERAGED OVER 1,000 TRIALS. THE IMPLIED OVERHEAD OF TINYSEC IS GIVEN AS THE SUM OF THE DATA'S MEANS. THE ROOT MEAN SQUARE IS DEFINED AS $\sqrt{s_{\text{w/o}}^2/1,000 + s_{\text{w/}}^2/1,000}$, WHERE $s_{\text{w/o}}$ AND $s_{\text{w/}}$ ARE THE DATA'S STANDARD DEVIATIONS.

	`encrypt()`	`computeMAC()`
Median	2,189 μsec	3,038 μsec
Mean	2,190 μsec	3,049 μsec
Standard Deviation	3 μsec	281 μsec
Standard Error	0 μsec	9 μsec

Implied Overhead of TinySec	5,239 μsec
Root Mean Square	9 μsec

Inasmuch as the goal at hand is distribution of 80-bit TinySec keys, any mechanism of exchange should provide at least as much security. According to NIST [18], then, the MICA2's implementation of Diffie-Hellman should employ a modulus, p, of at least 1,024 bits and an exponent (*i.e.*, private key), x, of at least 160 bits (Table V).

Unfortunately, on an 8-bit architecture, computations with 160-bit and 1,024-bit values are not inexpensive. However,

TABLE IV

MEMORY OVERHEAD OF TINYSEC, DETERMINED THROUGH INSTRUMENTATION OF CNTTORFM, AN APPLICATION WHICH SIMPLY BROADCASTS A COUNTER'S VALUES OVER THE MICA2'S RADIO. THE `.bss` AND `.data` SEGMENTS CONSUME SRAM WHILE THE `.text` SEGMENT CONSUMES ROM. STACK IS DEFINED HERE AS THE MAXIMUM OF THE APPLICATION'S STACK SIZE DURING EXECUTION.

	without TinySec	with TinySec	Difference
`.bss`	384 B	838	454 B
`.data`	4 B	280 B	276 B
`.text`	9,220 B	16,296 B	7,076 B
stack	105 B	197 B	92 B

modular exponentiation is not intractable on the MICA2. Fig. 3 offers the results of instrumentation of one implementation of Diffie-Hellman for the MICA2 [19]: computation of $2^x \pmod p$, where x is a pseudorandomly generated 160-bit integer and p is a 768-bit prime requires 31.0 sec on average; computation of the same, where p is a 1,024-bit prime, requires 54.9 sec. Assuming (generously) that nodes sharing some key need only be rekeyed every 2^{32} packets (at which time four-byte IVs are exhausted), this computation and that for $y^x \pmod p$, where y is another node's public key, seem reasonable costs for an application's longevity. Table VI details these operations' memory usage.

Of course, these measurements assume operation at full duty cycle, the energy requirements of which may be unacceptable, as the MICA2's lifetime decreases to just a few days at maximal duty cycle. Table VII reveals the MICA2's energy consumption for modular exponentiation: computation of $2^x \pmod p$ appears to require 1.185 J. Roughly speaking, a mote could devote its lifetime to 51,945 such computations.[3]

Of course, these numbers might be improved (with, *e.g.*, hand-optimization). Unfortunately, these computations require not only time but also memory. Mere storage of a public key requires as many bits as is the modulus in use. Accordingly, n 1,024-bit keys would more than exhaust a node's SRAM for n as small as 32. Although a node is unlikely to have—or, at least, need—so many neighbors or certificate authorities for whom it needs public keys, Diffie-Hellman's relatively large key sizes are unfortunate in the MICA2's resource-constrained environment. A key of this size would not even fit in a single TinyOS packet.

[3]For instance, Energizer No. E91, an AA battery, offers an average capacity of 2,850 mAh [20]; it follows that no more than 2 × 2,850 mAh × 3600 sec/h ÷ (7.3 mA × 54.1144 sec) ≈ 51,945 modular exponentiations would be possible with two AA batteries on the MICA2. Of course, this bound is generous: the MICA2 effectively dies once voltage drops below 2 volts.

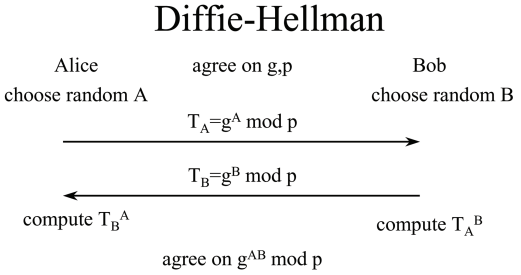

Fig. 2. Typical exchange of a shared secret under Diffie-Hellman based on DLP [21].

TABLE V

STRENGTH OF DIFFIE-HELLMAN BASED ON DLP FOR VARIOUS MODULI AND EXPONENTS. "AN ALGORITHM THAT HAS A 'Y' BIT KEY, BUT WHOSE STRENGTH IS EQUIVALENT TO AN 'X' BIT KEY OF SUCH A SYMMETRIC ALGORITHM IS SAID TO PROVIDE 'X BITS OF SECURITY' OR TO PROVIDE 'X-BITS OF STRENGTH'. AN ALGORITHM THAT PROVIDES X BITS OF STRENGTH WOULD, ON AVERAGE, TAKE $2^{X-1}T$ TO ATTACK, WHERE T IS THE AMOUNT OF TIME THAT IS REQUIRED TO PERFORM ONE ENCRYPTION OF A PLAINTEXT VALUE AND COMPARISON OF THE RESULT AGAINST THE CORRESPONDING CIPHERTEXT VALUE." [18]

Bits of Security	Modulus	Exponent
80	1,024	160
112	2,048	224
128	3,072	256
192	7,680	384
256	15,360	512

IV. ECDLP AND THE MICA2

With ECC, secure distribution of 80-bit TinySec keys is possible using public keys with fewer bits than 1,024: 163 bits are sufficient. Indeed, elliptic curves are believed to offer security computationally equivalent to that of Diffie-Hellman based on DLP with remarkably smaller key sizes insofar as subexponential algorithms exist for DLP [22]–[25], but no such algorithm is known or thought to exist for ECDLP over certain fields [26], [27].

Elliptic curves offer an alternative foundation for the exchange of shared secrets among eavesdroppers with perfect forward secrecy, as depicted in Fig. 4. ECDLP, on which ECC [28], [29] is based, typically involves recovery over some Galois (*i.e.*, finite) field, $\mathbb{F}$, of $k \in \mathbb{F}$, given (at least) $k \cdot G$, G, and E, where G is a point on an elliptic curve, E, a smooth curve of the long Weierstrass form

$$y^2 + a_1xy + a_3y \equiv x^3 + a_2x^2 + a_4x + a_6, \quad (1)$$

where $a_i \in \mathbb{F}$. Of recent interest to cryptographers are such curves over $\mathbb{F}_p$ and $\mathbb{F}_{2^p}$, where p is prime, as neither appears vulnerable to subexponential attack [27]. Though once popular,

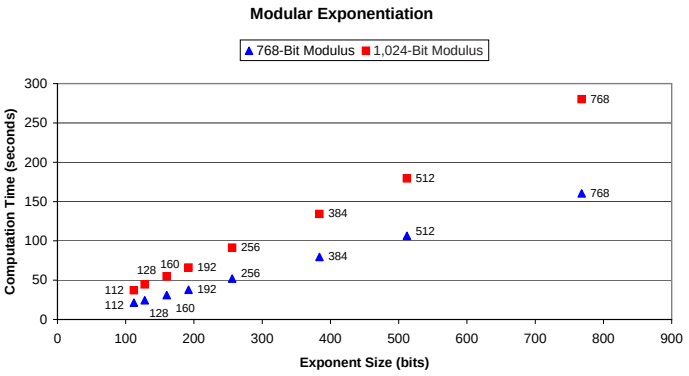

Fig. 3. Time required to compute $2^x \pmod p$, where p is prime, on the MICA2.

TABLE VI

MEMORY OVERHEAD OF MODULAR EXPONENTIATION, DETERMINED THROUGH INSTRUMENTATION OF AN IMPLEMENTATION OF DIFFIE-HELLMAN BASED ON DLP ON THE MICA2 WHICH COMPUTES $2^x \pmod p$, WHERE x IS A 512-BIT INTEGER AND p IS PRIME. THE `.bss` AND `.data` SEGMENTS CONSUME SRAM WHILE THE `.text` SEGMENT CONSUMES ROM. STACK IS DEFINED HERE AS THE MAXIMUM OF THE APPLICATION'S STACK SIZE DURING EXECUTION.

	768-Bit Modulus	1,024-Bit Modulus
`.bss`	852 B	980 B
`.data`	102 B	134 B
`.text`	11,334 B	11,350 B
stack	136 B	136 B

extension fields of composite degree over $\mathbb{F}_2$ are vulnerable by reduction with Weil descent [30] of ECDLP to DLP over hyperelliptic curves [27]. But $\mathbb{F}_{2^p}$, a binary extension field, remains popular among implementations of ECC, especially those in hardware, as it allows for particularly space- and time-efficient algorithms. In light of its applications in coding, the field has also received more attention in the literature than those of other characteristics [31].

It is with this history in mind that we proceeded with our implementation of ECC over $\mathbb{F}_{2^p}$ toward an end of smaller public keys for the MICA2.

A. *Elliptic Curves over* $\mathbb{F}_{2^p}$

It turns out that, over $\mathbb{F}_{2^p}$, Equation 1 simplifies to

$$y^2 + xy \equiv x^3 + ax^2 + b, \tag{2}$$

where $a, b \in \mathbb{F}_{2^p}$, upon substitution of $a_1^2 x + \frac{a_3}{a_1}$ for x and $a_1^3 y + \frac{a_1^2 a_4 + a_3^2}{a_1^3}$ for y, if we consider only nonsupersingular curves, for which $a_1 \neq 0$. It is the set of solutions to Equation 2 and, more generally, Equation 1 (*i.e.*, the points on E), that actually provides the foundation for smaller public keys on the

TABLE VII

ENERGY CONSUMPTION OF MODULAR EXPONENTIATION, DETERMINED THROUGH INSTRUMENTATION OF AN IMPLEMENTATION OF DIFFIE-HELLMAN BASED ON DLP ON THE MICA2 WHICH COMPUTES $2^x \pmod p$, WHERE x IS A 160-BIT INTEGER AND p IS A 1,024-BIT PRIME.

	1,024-Bit Modulus, 160-Bit Exponent
Total Time	54.1144 sec
Total CPU Utilization	3.9897×10^8 cycles
Total Energy	1.185 Joules

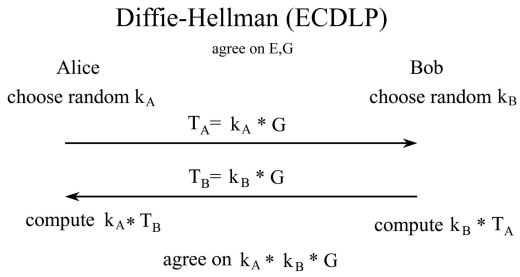

Fig. 4. Typical exchange of a shared secret under Diffie-Hellman based on ECDLP.

MICA2. All that remains is specification of some algebraic structure over that set. An Abelian group suffices but requires provision of some binary operator offering closure, associativity, identity, inversion, and commutativity. As suggested by ECDLP's definition, that operator is to be addition.

The addition of two points on a curve over $\mathbb{F}_{2^p}$ is defined as

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3),$$

such that

$$(x_3, y_3) = (\lambda^2 + \lambda + x_1 + x_2 + a,\ \lambda(x_1 + x_3) + x_3 + y_1),$$

where

$$\lambda = (y_1 + y_2)(x_1 + x_2)^{-1}.$$

However, so that the group is Abelian, it is necessary to define a "point at infinity," $\mathcal{O}$, whereby

$$\begin{aligned} \mathcal{O} + \mathcal{O} &= \mathcal{O}, \\ (x, y) + \mathcal{O} &= (x, y), \text{ and} \\ (x, y) + (x, -y) &= \mathcal{O}. \end{aligned}$$

Doubling of some point, meanwhile, is defined as

$$(x_1, y_1) + (x_1, y_1) = (x_3, y_3),$$

such that

$$(x_3, y_3) = (\lambda^2 + \lambda + a,\ x_1^2 + (\lambda + 1)x_3),$$

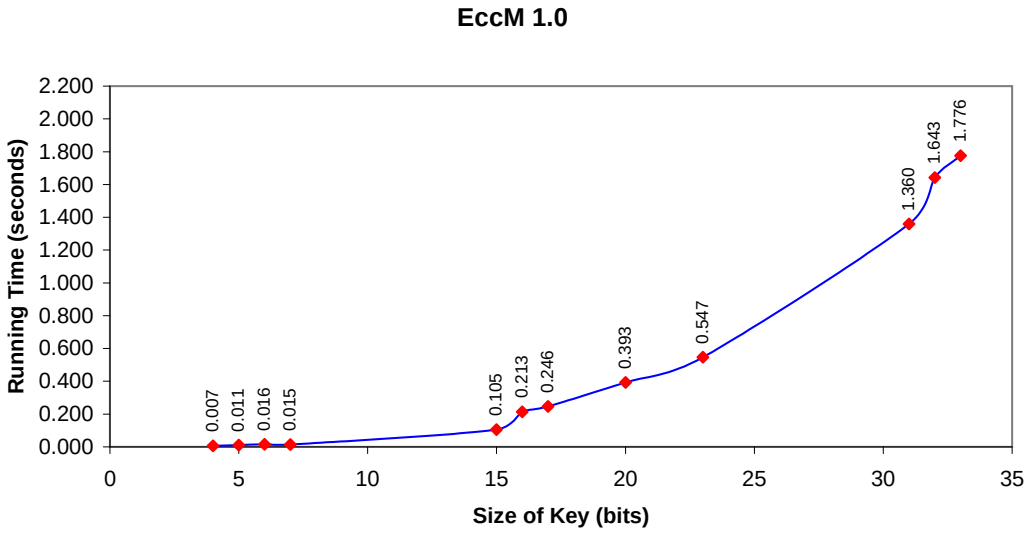

Fig. 5. Running time for EccM 1.0, a TinyOS module which selected for a node at random, using a polynomial basis over $\mathbb{F}_{2^p}$, a curve, a point, and a private key, thereafter computing the node's public key. Points are labelled with running times. For larger keys (*e.g.*, 63-bit), the module failed to produce results.

where

$$\lambda = x_1 + y_1 x_1^{-1},$$

provided $x_1 \neq 0$.

With these primitives is point multiplication also possible [32]. With an algebraic structure on the points of elliptic curves over $\mathbb{F}_{2^p}$ thus defined, implementation of a cryptosystem is possible.

B. ECC over $\mathbb{F}_{2^p}$

Implementation of ECC over $\mathbb{F}_{2^p}$ first requires a choice of basis for points' representation, insofar as each $a \in \mathbb{F}_{2^p}$ can be written as

$$a = \sum_{i=0}^{m-1} a_i \alpha_i,$$

where $a_i \in \{0, 1\}$. Thus defined, a can be represented as a binary vector, $\{a_0, a_1, \ldots, a_{p-1}\}$, where $\{\alpha_0, \alpha_1, \ldots, \alpha_{p-1}\}$ is its basis over $\mathbb{F}_2$. Most common for bases over $\mathbb{F}_2$ are polynomial bases and normal bases, though dual, triangular, and other bases exist.

When represented with a polynomial basis, each $a \in \mathbb{F}_{2^p}$ corresponds to a binary polynomial of degree less than p, whereby

$$a = a_{p-1}x^{p-1} + a_{p-2}x^{p-2} + \cdots + a_0 x^0,$$

where, again, $a_i \in \{0, 1\}$. Accordingly, each $a \in \mathbb{F}_{2^p}$ can be represented in the MICA2's SRAM as a bit string, $a_{p-1}a_{p-2}\cdots a_0$. All operations on these elements are performed modulo an irreducible reduction polynomial, f, of degree p over $\mathbb{F}_2$, such that $f(x) = x^p + \sum_{i=0}^{p-1} f_i x_i$, where $f_i \in \{0, 1\}$ for $i \in \{0, 1, \ldots, p-1\}$. Typically, if an irreducible trinomial, $x^p + x^k + 1$, exists over $\mathbb{F}_{2^p}$, then $f(x)$ is chosen to be that with smallest k; if no such trinomial exists, then $f(x)$ is chosen to be a pentanomial, $x^p + x^{k_3} + x^{k_2} + x^{k_1} + 1$, such that k_1 is minimal, k_2 is minimal given k_1, and k_3 is minimal given k_1 and k_2 [33].

In a polynomial basis, addition of two elements, a and b is defined as $a + b = c$, where $c_i \equiv a_i + b_i \pmod 2$ (*i.e.*, a sequence of XORs). Multiplication of a and b, meanwhile, is defined as $a \cdot b = c$, where $c(x) \equiv (\sum_{i=0}^{p-1} a_i x^i)(\sum_{i=0}^{p-1} b_i x^i) \pmod{f(x)}$.

We selected a polynomial basis for our implementations of point multiplication on the MICA2, as it tends to allow for more efficient implementations in software [34].

C. First Implementation

Our first implementation of ECC on the MICA2 (EccM 1.0), a TinyOS module based on code by Michael Rosing [35], whose *Implementing Elliptic Curve Cryptography* is a popular starting point for any implementation of ECC, ultimately reinforced prevailing wisdom: it was a failure.

EccM 1.0 first selected a random curve in the form of Equation 2, such that $a = 0$ and $b \in \mathbb{F}_{2^p}$. It next selected a random point, $G \in \mathbb{F}_{2^p} \times \mathbb{F}_{2^p}$, from that curve as well as a random $k \in \mathbb{F}_{2^p}$, the node's private key. Finally, it computed $k \cdot G$, the node's public key.

As in Rosing's code, this implementation employed a number of optimizations. Addition of points was implemented in accordance with Schroeppel *et al.* [36]; multiplication of points followed Koblitz [37]; conversion of integers to non-adjacent form was accomplished as in Solinas [38]. Generation

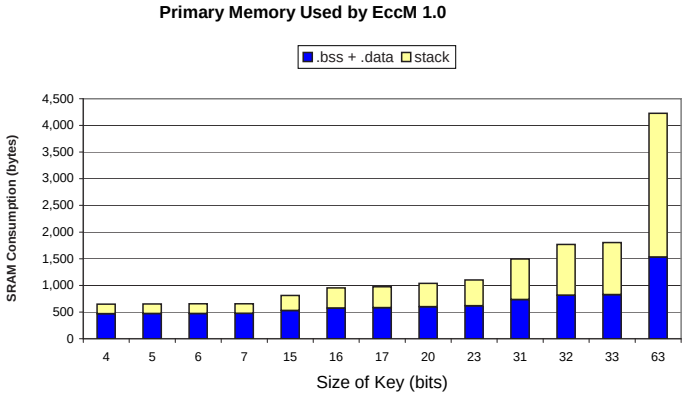

Fig. 6. Primary memory used by EccM 1.0, a TinyOS module which, using a polynomial basis over $\mathbb{F}_{2^p}$, selected for a node at random a curve, a point, and a private key, thereafter computing the node's public key. Although the sizes of the `.bss` and `.data` segments are fixed during execution, stack is defined here as the maximum of the application's stack size during execution. Keys of 63 bits or more exhaust the MICA2's 4,096 B of SRAM.

of pseudorandom numbers, meanwhile, was achieved with Marsaglia [39].

On first glance, the results (Fig. 5) were encouraging, with generation of 33-bit keys requiring just 1.776 sec. Unfortunately, for larger keys (*e.g.*, 63-bit), the module failed to produce results, instead causing the mote to reset as a result of stack overflow. Although none of the module's functions were recursive, several utilized a good deal of memory for multi-word arithmetic. Fig. 6 offers the results of an analysis of EccM 1.0's usage of SRAM.

D. Second Implementation

Since optimizations of EccM 1.0 failed to render generation of even 63-bit keys possible, an overhaul of this popular implementation proved necessary for realization of 163-bit keys. Inspired by the design of Dragongate Technologies Limited's Java-based jBorZoi 0.9 [40], EccM 2.0 similarly implements ECC but with far greater success. EccM 2.0 selects for a node, Alice, a private key, k_A, using a polynomial basis over $\mathbb{F}_{2^p}$, thereafter computing with a Koblitz curve and base point, G, Alice's public key, T_A. Alice's public key is then broadcasted (in two, 22-byte payloads) to any node, Bob, with whom secure communication is desired. Provided Alice receives Bob's public key, T_B, from Bob in this same manner, each can compute a shared secret, $k_A \cdot k_B \cdot G$, where k_B is Bob's private key. If so desired, this secret could be massaged into compliance with a standard like the Elliptic Curve Key Agreement Scheme, Diffie-Hellman 1 (ECKAS-DH1) [41].

In EccM 2.0, multiplication of points is achieved with Algorithm IV.1 in Blake *et al.* [42], while addition of points is achieved with Algorithm 7 in López and Dahab [33]. Multiplication of elements in $\mathbb{F}_{2^p}$, meanwhile, is implemented as Algorithm 4 in López and Dahab [43], while inversion is implemented as Algorithm 8 in Hankerson *et al.* [44].

Beyond rendering 163-bit public keys feasible, EccM 2.0 also redresses another shortcoming in EccM 1.0. Inasmuch as EccM 1.0 selects curves at random, it risks (albeit with exponentially small probability) selection of supersingular curves which are vulnerable to sub-exponential attack via MOV reduction [45] with index-calculus methods [46]. EccM 2.0 thus obeys NIST's recommendation for ECC over $\mathbb{F}_{2^p}$ [47], selecting, for the results herein,

$$f(x) = x^{163} + x^7 + x^6 + x^3 + 1$$

for the reduction polynomial,

$$y^2 + xy \equiv x^3 + x^2 + 1$$

for the curve, E, the order of (*i.e.*, number of points on) which is 0x40000000000000000000020108a2e0cc0d99f8a5ef, and, for the point $G = (G_x, G_y)$,

$$G_x = \text{0x2fe13c0537bbc11acaa07d793de4e6d5e5c94eee8}$$

and

$$G_y = \text{0x289070fb05d38ff58321f2e800536d538ccdaa3d9.}$$

Ultimately, not only does EccM 2.0 employ much less memory than does EccM 1.0 (Table VIII), its running time bests that for Diffie-Hellman based on DLP, using keys an order of magnitude smaller in size but no less secure. The time required to generate a private and public key pair with this module, averaged over 100 trials, is just 34.161 sec, with a standard deviation of 0.921 sec. The time required to generate a shared secret, given one's private key and another's public key, averaged over 100 trials, is 34.173 sec, with a standard deviation of 0.934 sec. In short, distribution of some shared secret involves just over one minute of computation per node in total. Table IX details the module's energy consumption. Although such performance might prove unacceptable for some applications of PKI, it appears quite reasonable for infrequent distribution of TinySec keys.

A link to EccM 2.0's source code is offered in the Appendix.

V. Discussion

EccM 2.0's average running time of, roughly, 34 seconds for point multiplication is the result of several iterations of optimization. In fact, this module initially clocked 7.782 minutes for this computation, well beyond any reasonable bound. To be sure, we spent some cycles foolishly (*e.g.*, unnecessarily recomputing the terminal condition for some loop). But other waste was less obvious. Apparent only to us (and not to NesC's compiler), certain loops were simply better off iterating from high to low than from low to high, given the expected lengths

TABLE VIII

MEMORY USAGE OF ECCM 1.0 VERSUS ECCM 2.0. WITH ECCM 2.0, WE OBTAIN SIGNIFICANTLY MORE BITS OF SECURITY USING A REASONABLE FOOTPRINT IN MEMORY. THE `.bss` AND `.data` SEGMENTS CONSUME SRAM WHILE THE `.text` SEGMENT CONSUMES ROM. STACK IS DEFINED HERE AS THE MAXIMUM OF THE APPLICATION'S STACK SIZE DURING EXECUTION. MUCH OF THE INCREASE OF ROM'S CONSUMPTION IS THE RESULT OF ECCM 2.0'S ADDITIONAL FUNCTIONALITY.

	EccM 1.0 (32-bit key)	EccM 2.0 (163-bit key)
`.bss`	826 B	1,055 B
`.data`	6 B	4 B
`.text`	17,544 B	34,342 B
stack	976 B	81 B

TABLE IX

ENERGY CONSUMPTION OF ECCM 2.0, A TINYOS MODULE WHICH ALLOWS TWO NODES TO GENERATE PUBLIC AND PRIVATE KEYS (AND, THEREAFTER, TO USE THE SAME TO EXCHANGE A SHARED SECRET), DURING GENERATION OF A NODE'S PUBLIC AND PRIVATE KEYS.

	Private-Key Generation	Public-Key Generation
Total Time	0.229 sec	34.161 sec
Total CPU Utilization	1.690×10^6 cycles	2.512×10^8 cycles
Total Energy	0.00549 Joules	0.816 Joules

of various multi-precision intermediates. Other loops proved better off once manually unrolled.

Rather than handle multi-precision bit shifts with a generalized implementation, we were able to shave seconds off the running time by special-casing the most common of shifts (namely left shifts by one bit and by two bits), albeit at a cost of a larger `.bss` segment.

Consider that, with inlining disabled, even the current version of this module induces hundreds of thousands of function calls, largely the result of the module's requirement of multi-precision arithmetic. Even the slightest of improvements in some function's performance, then, can effect significant gains overall.

Of course, some optimizations were grounded in published, theoretical results. Substitution of Algorithm 2 in Hankerson *et al.* [44] with Algorithm 4 in López and Dahab [43] offered several seconds of improvement, as did implementation of Algorithm 7 in López and Dahab [33]. But the art of source-level, hand optimizations, so infrequently deployed for modern systems, proved remarkably helpful, daresay necessary, for an environment so constrained as the MICA2.

VI. FUTURE WORK

While ECC's performance on the MICA2 is gratifying, opportunities for future work remain. Further reduction of the module's running time, through source- or assembly-level enhancements, is, of course, of interest. Worthy of consideration for future versions of this module is a normal basis, an advantage of which would be its implementation using only ANDs, XORs, and cyclic shifts, beneficiaries of which are multiplication and squaring. (For this reason, normal bases tend to be popular in implementations of ECC in hardware.) Of value as well might be a hybrid of polynomial and normal bases, as such is thought to leverage advantages of each simultaneously [35].

Of course, recent work by Gura *et al.* [11] suggests that the module might offer even better performance if re-implemented over $\mathbb{F}_p$, especially as expensive inversions could be avoided through use of projective (as opposed to affine) coordinates [48]. Although relatively efficient algorithms exist for modular reduction (*e.g.*, those of Montogomery [49] or Barrett [50]), selection of a generalized Mersene number for p would also allow modular reduction to be executed as a more efficient sequence of three additions (mod p) [51].

Performance aside, EccM 2.0's reliance on TinyOS's RandomLFSR module is troubling cryptographically, as this pseudo-random number generator (PRNG) relies solely upon a mote's unique ID for seeding, rather than upon any physical source of randomness. Implementation of a superior PRNG is necessary for our module's security. Truly random bits might be captured from such sources as local sensor readings, interrupt and packet-arrival times, and other physical sources.

VII. RELATED WORK

Studied by mathematicians for more than a century, elliptic curves claim significant coverage in the literature. ECC, meanwhile, has received much attention since its discovery in 1985.

Of particular relevance to this work is Woodbury's recommendation of an optimal extension field, $\mathbb{F}_{(2^8-17)^{17}}$, for low-end, 8-bit processors [52]. Jung *et al.* propose supplementary hardware for AVR implementing operations over binary fields [53]. Handschuh and Paillier propose cryptographic co-processors for smart cards [54], whereas Woodbury *et al.* describe ECC for smart cards without coprocessors [55]. Albeit for a different target, Hasegawa *et al.* provide a "small and fast" implementation of ECC in software over $\mathbb{F}_p$ for a 16-bit microcomputer [56]. Messerges *et al.* call for ECC with 163-bit keys for mobile, *ad hoc* networks [57]. Guajardo *et al.* describe an implementation of ECC for the 16-bit TI MSP430x33x family of microcontrollers [58]. Weimerskirch *et al.*, meanwhile, offer an implementation of ECC for Palm OS [59], and Brown *et al.* offer the same for Research In Motion's RIM pager [60].

ZigBee, on the other hand, shares this work's aim of wireless security for sensor networks albeit not with ECC but with AES-128 [61], a shared-key protocol. Meanwhile, recommendations for ECC's parameters abound, among academics [62], among corporations [63], and within government [41], [47].

A number of implementations of ECC in software are freely available, though none are particularly well-suited for the MICA2, in no small part because of their memory requirements. Rosing [35] offers his C-based implementation of ECC over $\mathbb{F}_{2^p}$ with both polynomial and normal bases. ECC-LIB [64] and pegwit [65] offer their own C-based implementations over $\mathbb{F}_{2^p}$ with polynomial bases. MIRACL [66] provides the same, with an additional option for curves over $\mathbb{F}_p$. LibTomCrypt [67], also in C, focuses on $\mathbb{F}_p$. Dragongate Technologies Limited, meanwhile, offers borZoi and jBorZoi [40], implementations of ECC over $\mathbb{F}_{2^p}$ with polynomial bases in C++ and Java, respectively. Another implementation in C++, also using a polynomial basis over $\mathbb{F}_{2^p}$, is available through libecc [68].

VIII. Conclusion

Despite claims to the contrary, public-key infrastructure appears viable on the MICA2, certainly for infrequent distribution of shared secrets. Although our implementation of ECC in 4 KB of primary memory on this 8-bit, 7.3828-MHz device offers room for further optimization, even a minute's worth of computation every 2^{32} transmissions (or every day or every week) seems reasonable for re-keying.

The need for PKI's success on the MICA2 seems clear. TinySec's shared secrets do allow for efficient, secure communications among nodes. But such devices as those in sensor networks, for which physical security is unlikely, require some mechanism for secret keys' distribution.

In that it offers equivalent security at lower cost to memory and bandwidth than does Diffie-Hellman based on DLP, a public-key infrastructure for key distribution based on elliptic curves is an apt, and viable, choice for TinyOS on the MICA2.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 310877. Many thanks to Breanne Duncan, Mark Hempstead, Michael Mitzenmacher, and Victor Shnayder of Harvard University for their assistance with this work.

Appendix

EccM 2.0's source code is available for download from `http://www.eecs.harvard.edu/~malan/`.

References

[1] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao, "Habitat monitoring: Application Driver for Wireless Communications Technology," 2001.

[2] V. A. Kottapalli, A. S. Kiremidjian, J. P. Lynch, E. Carryer, T. W. Kenny, K. H. Law, and Y. Lei, "Two-tiered wireless sensor network architecture for structural health monitoring," SPIE's 10th Annual International Symposium on Smart Structures and Materials, March 2003.

[3] Vital Dust: Wireless Sensor Networks for Emergency Medical Care, http://www.eecs.harvard.edu/~mdw/proj/vitaldust/.

[4] NEST Challenge Architecture, August 2002.

[5] I. Crossbow Technology, "MICA2: Wireless Measurement System," http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0042-04_A_MICA2.pdf.

[6] C. Karlof, N. Sastry, and D. Wagner, "TinySec: Link Layer Security for Tiny Devices," http://www.cs.berkeley.edu/~nks/tinysec/.

[7] A. Perrig, J. Stankovic, and D. Wagner, "Security in Wireless Sensor Networks," *Communications of the ACM*, vol. 47, no. 6, pp. 53–57, June 2004.

[8] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar, "SPINS: Security Protocols for Sensor Networks," in *Mobile Computing and Networking*, 2001, pp. 189–199.

[9] C. Karlof, N. Sastry, and D. Wagner, "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks." Baltimore, Maryland: Second ACM Conference on Embedded Networked Sensor Systems, November 2004.

[10] R. Watro, "Lightweight Security for Wireless Networks of Embedded Systems," http://www.is.bbn.com/projects/lws-nest/bbn_nest_apr_03.ppt, May 2003.

[11] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs." Boston, Massachusetts: 6th International Workshop on Cryptographic Hardware and Embedded Systems, August 2004.

[12] Computer Security Division, *SKIPJACK and KEA Algorithm Specifications*, National Institute of Standards and Technology, May 1988.

[13] National Institute of Standards and Technology, "Federal Information Processing Standards Publication 185: Escrowed Encryption Standard (EES)," February 1994. [Online]. Available: {http://www.itl.nist.gov/fipspubs/fip185.htm}

[14] E. Biham, A. Biryukov, and A. Shamir, "Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials," *Lecture Notes in Computer Science*, vol. 1592, pp. 12–23, 1999. [Online]. Available: citeseer.nj.nec.com/biham99cryptanalysis.html

[15] Naveen Sastry, University of California at Berkeley, personal correspondence.

[16] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, no. 6, pp. 644–654, 1976. [Online]. Available: citeseer.nj.nec.com/diffie76new.html

[17] W. Diffie, P. C. van Oorschot, and M. J. Wiener, "Authentication and authenticated key exchanges," *Designs, Codes, and Cryptography*, vol. 2, no. 2, pp. 107–125, 1992.

[18] National Institute of Standards and Technology, "Special Publication 800-57: Recommendation for Key Management," January 2003. [Online]. Available: {http://csrc.nist.gov/CryptoToolkit/kms/guideline-1-Jan03.pdf}

[19] BBN Technologies, "Diffie-Hellman 1," July 2003.

[20] Everyready Battery Company, "Engineering Datasheet: Energizer No. X91," http://data.energizer.com/datasheets/library/primary/alkaline/energizer/consumer_oem/e91.pdf.

[21] H. U. Radia Perlman, Computer Science 243.

[22] L. M. Adleman, "A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography," in *Proc. 20th IEEE Found. Comp. Sci. Symp.*, 1979, pp. 55–60.

[23] D. M. Gordon, "Discrete Logarithms in GF(P) Using the Number Field Sieve," *SIAM J. Discret. Math.*, vol. 6, no. 1, pp. 124–138, 1993.

[24] B. A. LaMacchia and A. M. Odlyzko, "Computation of Discrete Logarithms in Prime Fields," *Lecture Notes in Computer Science*, vol. 537, pp. 616–618, 1991. [Online]. Available: citeseer.nj.nec.com/lamacchia91computation.html

[25] M. Rabin, "Digitalized signatures and public-key functions as intractable as factorization," MIT, Tech. Rep. MIT/LCS/TR-212, 1979.

[26] Certicom Corporation, "Remarks On The Security Of The Elliptic Curve Cryptosystem," http://www.comms.engg.susx.ac.uk/fft/crypto/EccWhite3.pdf, July 2000.

[27] P. Gaudry, F. Hess, and N. P. Smart, "Constructive and Destructive Facets of Weil Descent on Elliptic Curves," tech-reports/2000/2000-gaudry.ps.gz, Department of Computer Science, University of Bristol, Tech. Rep. CSTR-00-016, October 2000.

[28] N. Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.

[29] V. Miller, "Uses of Elliptic Curves in Cryptography," in *Lecture Notes in Computer Science 218: Advances in Crytology - CRYPTO '85*. Berlin: Springer-Verlag, 1986, pp. 417–426.

[30] G. Frey and H. Gangl, "How to Disguise an Elliptic Curve (Weil Descent)," ECC '98, September 1998.

[31] C. Paar, "Implementation Options for Finite Field Arithmetic for Elliptic Curve Cryptosystems," Invited presentation at the 3rd Workshop on Elliptic Curve Cryptography (ECC '99), November 1999.

[32] D. M. Gordon, "A Survey of Fast Exponentiation Methods," *J. Algorithms*, vol. 27, no. 1, pp. 129–146, 1998. [Online]. Available: citeseer.nj.nec.com/gordon97survey.html

[33] J. López and R. Dahab, "An Overview of Elliptic Curve Cryptography," Institute of Computing, Sate University of Campinas, São Paulo, Brazil, Tech. Rep., May 2000.

[34] G. Barwood, "Elliptic Curve Cryptography FAQ v1.12 22nd," http://www.cryptoman.com/elliptic.htm, December 1997.

[35] M. Rosing, *Implementing Elliptic Curve Cryptography*, K. Antonsen, Ed. Manning Publications Co., 1999.

[36] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck, "Fast Key Exchange with Elliptic Curve Systems," *Lecture Notes in Computer Science*, vol. 963, 1995. [Online]. Available: citeseer.nj.nec.com/schroeppel95fast.html

[37] N. Koblitz, "CM-curves with good cryptographic properties," in *Advances in Cryptology – CRYPTO '91*, 1992, pp. 279–287.

[38] J. A. Solinas, "An Improved Algorithm for Arithmetic on a Family of Elliptic Curves," in *Advances in Cryptology – CRYPTO '97*, 1997, pp. 357–371.

[39] G. Marsaglia, "The Mother of All Random Generators," ftp://ftp.taygeta.com/pub/c/mother.c, October 1994.

[40] Dragongate Technologies Limited, "jBorZoi 0.9," http://dragongate-technologies.com/products.html, August 2003.

[41] Microprocessor and M. S. Committee, *IEEE Standard Specifications for Public-Key Cryptography*, IEEE Computer Society, January 2000.

[42] I. Blake, G. Seroussi, and N. Smart, "Elliptic Curves in Cryptography," *LMS Lecture Note Series*, vol. 265, 1999.

[43] J. López and R. Dahab, "High-Speed Software Multiplication in $\mathbb{F}_{2^m}$," Institute of Computing, Sate University of Campinas, São Paulo, Brazil, Tech. Rep., May 2000.

[44] D. Hankerson, J. L. Hernandez, and A. Menezes, "Software Implementation of Elliptic Curve Cryptography over Binary Fields," *Lecture Notes in Computer Science*, vol. 1965, 2001. [Online]. Available: citeseer.nj.nec.com/hankerson00software.html

[45] A. Menezes, S. Vanstone, and T. Okamoto, "Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field," in *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. ACM Press, 1991, pp. 80–89.

[46] J. Silverman and J. Suzuki, "Elliptic Curve Discrete Logarithms and the Index Calculus," in *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology*. LNCS, Springer-Verlag, 1998.

[47] National Institute of Standards and Technology, "Recommended Elliptic Curves For Federal Government Use," http://csrc.nist.gov/CryptoToolkit/dss/ecdsa/NISTReCur.pdf, July 1999.

[48] J. Grosschädl, "Implementation Options for Elliptic Curve Cryptography," http://www.iaik.tugraz.at/teaching/02_it-sicherheit/04_vortragsthemen/ECC.pdf, April 2003.

[49] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[50] P. Barrett, "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor," in *Advances in Cryptology – CRYPTO '86*, A. M. Odlyzko, Ed., vol. 263, 1987.

[51] J. Solinas, "Generalized mersenne numbers," citeseer.nj.nec.com/solinas99generalized.html, University of Waterloo, Tech. Rep. CORR-39, 1999.

[52] A. D. Woodbury, "Efficient Algorithms for Elliptic Curve Cryptosystems on Embedded Systems," http://www.wpi.edu/Pubs/ETD/Available/etd-1001101-195321/unrestricted/woodbury.pdf, September 2001.

[53] M. Jung, M. Ernst, F. Madlener, S. Huss, and R. Blümel, "A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2^n)," http://ece.gmu.edu/crypto/ches02/talks_files/Jung.ppt.

[54] H. Handschuh and P. Paillier, "Smart Card Crypto-Coprocessors for Public-Key Cryptography," in *Lecture Notes in Computer Science*, ser. Smart Card Research and Applications, J.-J. Quisquater and B. Schneier, Eds. Springer-Verlag, 2000, pp. 386–394.

[55] A. D. Woodbury, D. V. Bailey, and C. Paar, "Elliptic Curve Cryptography On Smart Cards without Coprocessors." Bristol, UK: The Fourth Smart Card Research and Advanced Applications (CARDIS 2000) Conference, September 2000.

[56] T. Hasegawa, J. Nakajima, and M. Matsui, "A Small and Fast Software Implementation of Elliptic Curve Cryptosystems over GF(p) on a 16-Bit Microcomputer," *IEICE Trans. Fundamentals*, vol. E82-A, no. 1, pp. 98–106, January 1999.

[57] T. S. Messerges, J. Cukier, T. A. M. Kevenaar, L. Puhl, R. Struik, and E. Callaway, "A Security Design for a General Purpose, Self-Organizing, Multihop Ad Hoc Wireless Network." George Mason University, Fairfax, Virginia: ACM Workshop on Security of Ad Hoc and Sensor Networks, October 2003.

[58] J. Guajardo, R. Blümel, U. Krieger, and C. Paar, "Efficient Implementation of Elliptic Curve Cryptosystems on the TI MSP430x33x Family of Microcontrollers," in *PKC 2001*, K. Kim, Ed., Korea, 2001, pp. 365–382.

[59] A. Weimerskirch, C. Paar, and S. C. Shantz, "Elliptic Curve Cryptography on a Palm OS Device." Sydney, Australia: The 6th Australasian Conference on Information Security and Privacy, July 2001.

[60] M. Brown, D. Cheung, D. Hankerson, J. L. Hernandez, M. Kirkup, and A. Menezes, "PGP in Constrained Wireless Devices," in *Proceedings of the 9th USENIX Security Symposium*. USENIX Association, August 2000.

[61] ZigBee Alliance, http://www.zigbee.org/.

[62] A. K. Lenstra and E. R. Verheul, "Selecting Cryptographic Key Sizes," October 1999.

[63] Certicom Corporation, "Standards for Efficient Cryptography Group," http://www.secg.org/.

[64] C. Zaroliagis, "ECC-LIB: A Library for Elliptic Curve Cryptography," http://www.ceid.upatras.gr/faculty/zaro/software/ecc-lib/.

[65] pegwit, http://groups.yahoo.com/group/pegwit/files/.

[66] S. S. Ltd, "Multiprecision Integer and Rational Arithmetic C/C++ Library," http://indigo.ie/~mscott/#Elliptic.

[67] T. S. Denis, "LibTomCrypt," http://libtomcrypt.org/.

[68] libecc, http://libecc.howpublishedforge.net/.