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

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Published Version</td>
<td><a href="http://dx.doi.org/10.1109/SAHCN.2004.1381904">http://dx.doi.org/10.1109/SAHCN.2004.1381904</a></td>
</tr>
<tr>
<td>Citable link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:2829933">http://nrs.harvard.edu/urn-3:HUL.InstRepos:2829933</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
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 $\mathbb{F}_2$ 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 [11]. 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]).\footnote{Although TinySec allows for 80-bit keys, its current implementation actually relies on 64-bit keys that are extended with 16 bits of padding.} 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 ($\mu$sec) on average, as suggested by transmission times, is nearly subsumed by the data’s root mean square (1,094 $\mu$sec). 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 $\mu$sec on average, and computation of that payload’s MAC requires 3,049 $\mu$sec on average; overall, TinySec adds 5,239 ± 18 $\mu$sec 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.\footnote{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.} 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

<table>
<thead>
<tr>
<th>Without TinySec</th>
<th>With TinySec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median</td>
<td>72,904 $\mu$sec</td>
</tr>
<tr>
<td>Mean</td>
<td>74,844 $\mu$sec</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>24,248 $\mu$sec</td>
</tr>
<tr>
<td>Standard Error</td>
<td>767 $\mu$sec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implied Overhead of TinySec</th>
<th>Root Mean Square</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,244 $\mu$sec</td>
<td>1,094 $\mu$sec</td>
</tr>
</tbody>
</table>

**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 difficulty 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
thoughr is the cost of such establishment on the x's implementation of Diffie-Hellman should employ 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.

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.

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,
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.5

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.

5For instance, Energizer No. E91, an AA battery, offers an average capacity of 2,850 mAh [20]; it follows that no more than $2 \times 2,850$ mAh $\times 3600$ sec $\div (7.3$ mAh $\times 54,1144$ sec) $\approx 51,945$ 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.

### 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 $k \cdot G$, $G$, and $E$, where $G$ is a point on an elliptic curve, $E$, a smooth curve of the long Weierstrass form

$$y^2 + a_1 xy + a_3 y \equiv x^3 + a_2 x^2 + a_4 x + a_6,$$

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

$$y^2 + xy = x^3 + ax^2 + b,$$

where $a, b \in \mathbb{F}_{2^p}$, upon substitution of $a_1^2x + a_1x$ for $x$ and $a_1^3 + a_1^2a_4 + a_5$ 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 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

$$(\mathcal{O} + \mathcal{O}) = \mathcal{O},$$

$$(x, y) + \mathcal{O} = (x, y),$$

and

$$(x, y) + (x, -y) = \mathcal{O}.$$
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 Schroeppe1 et al. [36]; multiplication of points followed Koblitz [37]; conversion of integers to non-adjacent form was accomplished as in Solinas [38]. Generation
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 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.

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,

\[
y^2 + x y \equiv x^3 + x^2 + 1
\]

for the curve, \( E \), the order of (i.e., number of points on) which is 0x4000000000000000000000020108a2e0cc0d99f8a5ef, and, for the point \( G = (G_x, G_y) \),

\[
G_x = 0x2f1e130537bb11aaca07d793de4e6d5e5c94ee8
\]

and

\[
G_y = 0x289070fb05d38ff58321f2e800536d538ccdaa3d9.
\]

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

![Primary Memory Used by EccM 1.0](image-url)
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.

<table>
<thead>
<tr>
<th></th>
<th>EccM 1.0 (32-bit key)</th>
<th>EccM 2.0 (163-bit key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bss</td>
<td>826 B</td>
<td>1,055 B</td>
</tr>
<tr>
<td>.data</td>
<td>6 B</td>
<td>4 B</td>
</tr>
<tr>
<td>.text</td>
<td>17,544 B</td>
<td>34,342 B</td>
</tr>
<tr>
<td>stack</td>
<td>976 B</td>
<td>81 B</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th></th>
<th>Private-Key Generation</th>
<th>Public-Key Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>0.229 sec</td>
<td>34.161 sec</td>
</tr>
<tr>
<td>Total CPU Utilization</td>
<td>1.690 x 10^8 cycles</td>
<td>2.512 x 10^8 cycles</td>
</tr>
<tr>
<td>Total Energy</td>
<td>0.00549 Joules</td>
<td>0.816 Joules</td>
</tr>
</tbody>
</table>

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 Montgomery [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, \(F_{(2^{17}−17)}\), for low-end, 8-bit processors [52]. Jung et al. propose supplementary hardware for AVR implementing operations over binary fields [53]. Handschu and Paillier propose cryptographic coprocessors 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. MIRAClark [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


