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

Qualifying Examination

David J. Malan
Division of Engineering and Applied Sciences
Harvard University

16 November 2004

MIT-EMS

Motivation for this work

CodeBlue

Managing *n* patients, for *n* = 1, is (relatively) easy

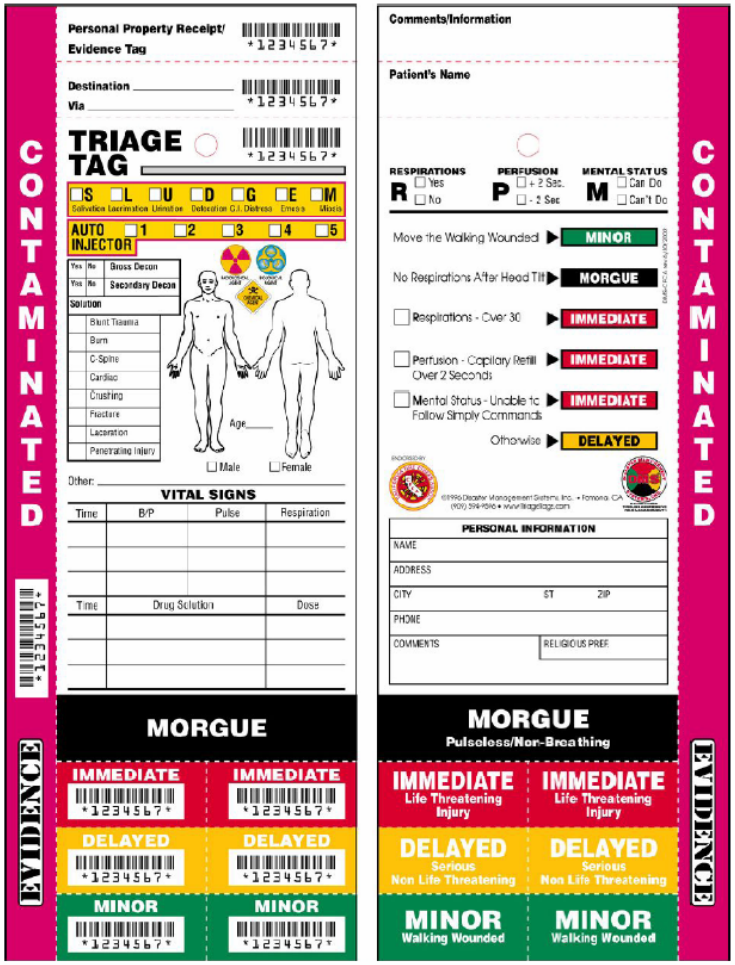
Image exerpted from "Vital Signs and Patient History," Eastern Medical Educators, 2000.

CodeBlue

Managing *n* patients, for large *n*, is much harder

CodeBlue

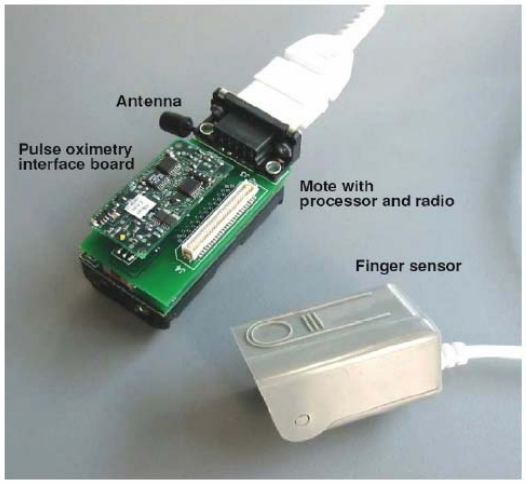
Current systems for monitoring patients are radio-, phone-, and paper-based

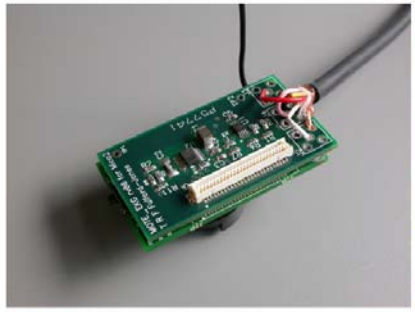


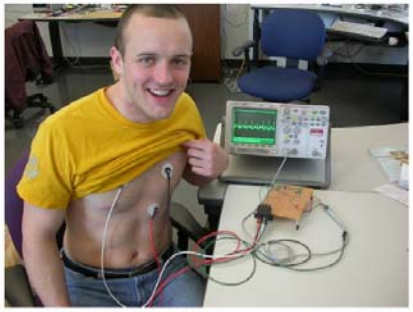
CodeBlue

An *Ad Hoc* Sensor Network Infrastructure for Emergency Medical Care

- Sensor networks (*e.g.*, mote-based pulse oximeters, EKGs) have potential for large impact
  - Real-time, continuous monitoring
  - Immediate alerts of changes in patient status
  - Electronic triage tags to store patient data
  - Relay data to hospital, correlate with patient records
- Transmission of patients' private data must be secure
  - Health Insurance Portability and Accountability Act of 1996 (HIPAA)







Ensuring Privacy and Security

Secret-key cryptography: an obvious solution?

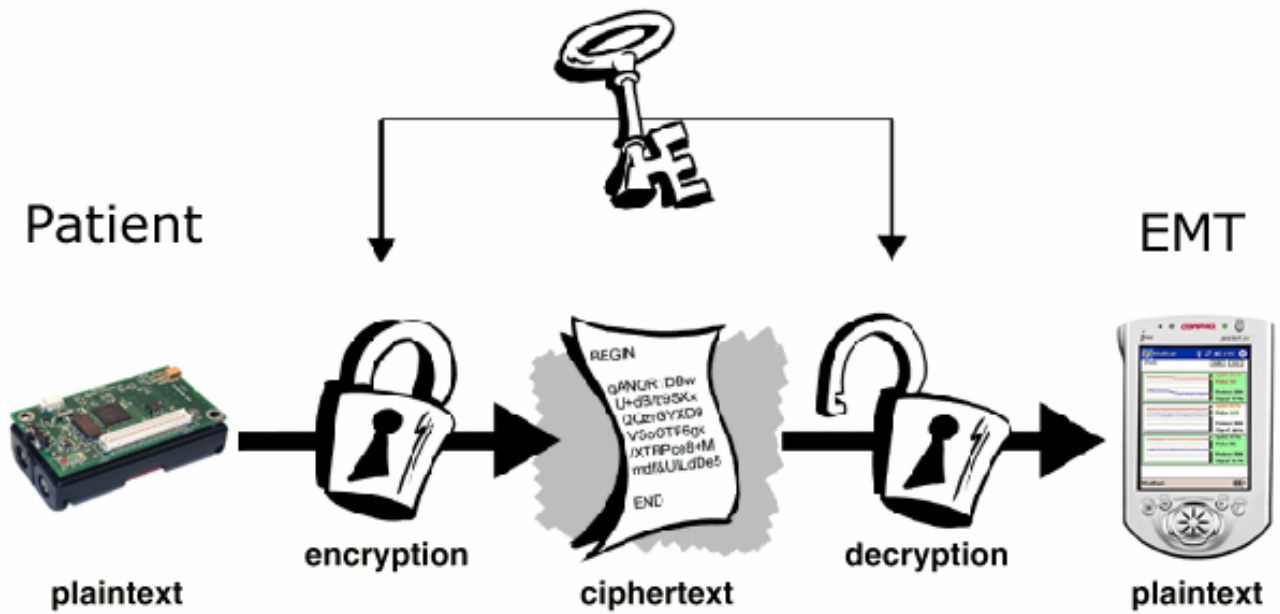
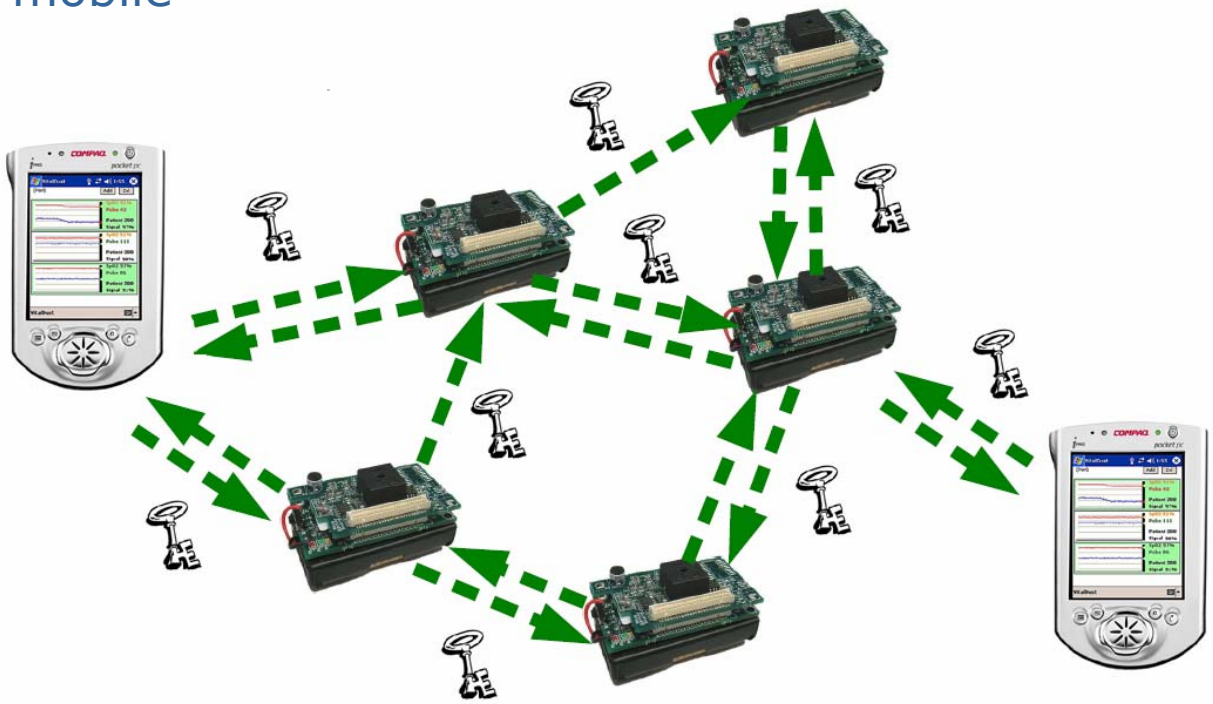


Image adapted from http://www.nuitari.de/crypto.html.

Ensuring Privacy and Security

The problem: key distribution

- How to establish shared secrets among authorized nodes?
  - Motes themselves are vulnerable to theft
  - Network participants are coming and going
  - Nodes are mobile



Ensuring Privacy and Security

Coordination among ambulance companies is non-trivial

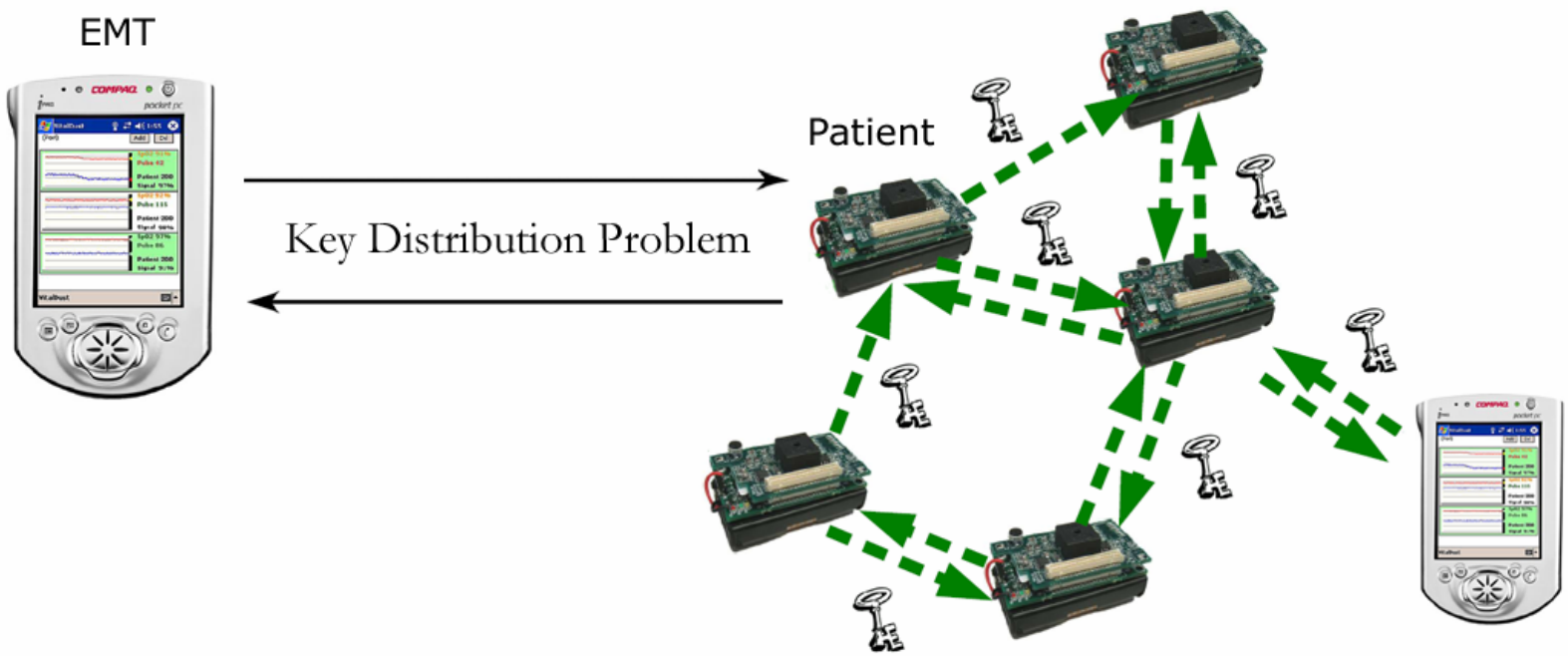
Acton Fire Dept. • American Medical Response of Mass. Inc. • Arlington Fire Department • Armstrong Ambulance Svc. • Ashland Fire Dept. • Marlboro Hudson Amb & Wheelchair Svc., Inc. • Bedford Fire Dept. • Belmont Fire Dept. • Boston EMS • Boston Med Flight • Boxborough Fire Dept. • Cambridge Fire Dept. • Canton Fire Dept. • Cohasset Fire Department • Concord Fire Dept. • Burlington Fire Dept. • Carlisle Fire Dept. • Dover Fire Dept. • Eastern Emergency Med. Svcs. • Fallon Ambulance Svc. • General Ambulance Svc., Inc. • Hanover Fire Dept. • Hingham Fire & Rescue Svcs. • Holliston Fire Dept. • Hopkinton Fire Dept. • Hudson Fire Dept. • Hull Fire/Rescue & Emergency Svcs. • Lexington Fire Dept. • Littleton Fire Dept. • MIT-EMS • Cataldo Ambulance Svc. • Maynard Fire Dept. • Medfield Fire Dept. • Millis Ambulance (Town of) • Natick Fire Dept. • Needham Fire Department • Norfolk Ambulance Svc. • Northeastern University Public Safety • Norwell Fire Department • Norwood Fire Dept. Ambulance • Professional Ambulance and Oxygen Service • Sharon Ambulance Svc. • Sherborn Fire Dept. Rescue Squad • Spaulding Rehabilitation Hospital Ambulance • Lincoln Ambulance Svc. (Town of) • Stow Emergency Med. Svc. (Town of) • Sudbury Fire Dept. • Walpole Fire & Rescue Dept. • Watertown Fire Dept. • Wayland Fire Dept. • Weston Fire Dept. • Westwood Fire Dept. • Wilmington Fire Dept. • Winchester Fire Dept. Rescue • Wrentham Fire Dept. • Pathways Ambulance Svc., Ltd. • Emerson Hospital EMS • Eas Care Ambulance • Caritas Norwood Hospital EMS • South Shore Hospital • Scituate Fire Dept. • Southborough Fire Dept. • Woburn Fire Dept. • Care Ambulance Svc. • Children's Hospital Boston • Wellesley Fire Department • Commonwealth Ambulance and Emergency Medical Services

List generated with `http://db.state.ma.us/dph/amb/amb_search.asp`.

Ensuring Privacy and Security

Ad hoc configuration is crucial

- How to establish a shared secret between EMT and patient?
- How to authenticate EMT?
- How to enable perfect forward secrecy?



Ensuring Privacy and Security

A solution: public key infrastructure (PKI)

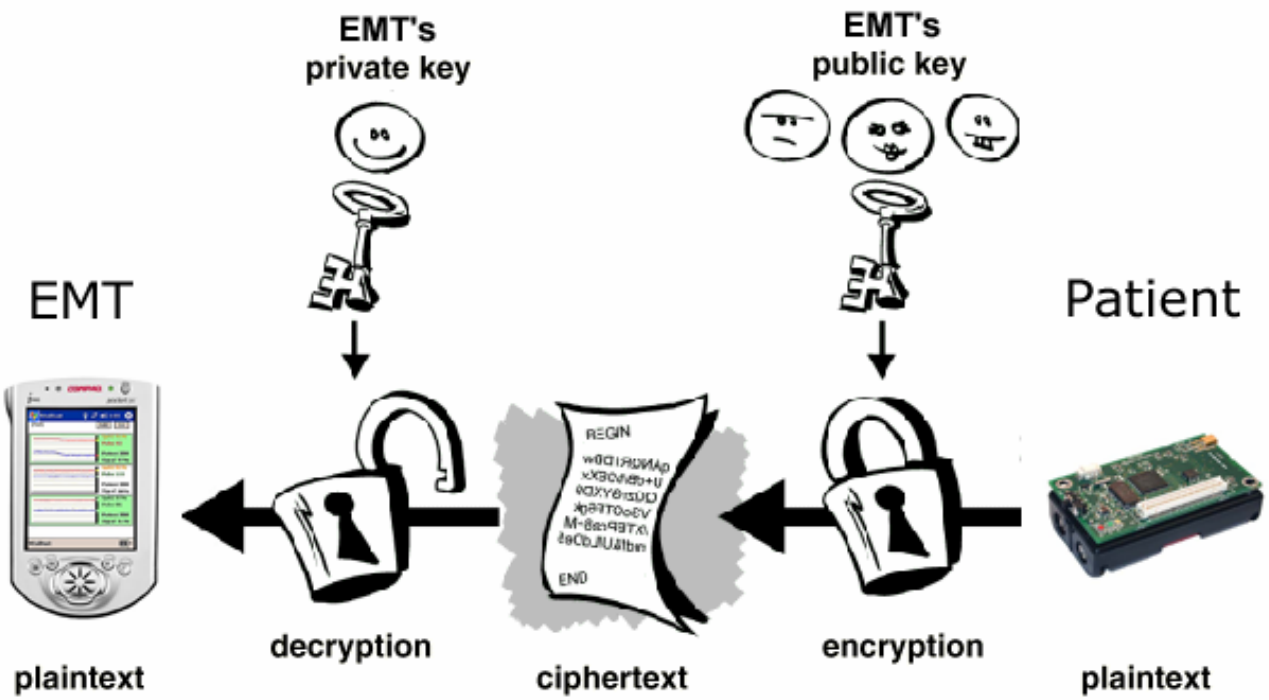


Image adapted from http://www.nuitari.de/crypto.html.

Ensuring Privacy and Security

A solution: public key infrastructure (PKI)

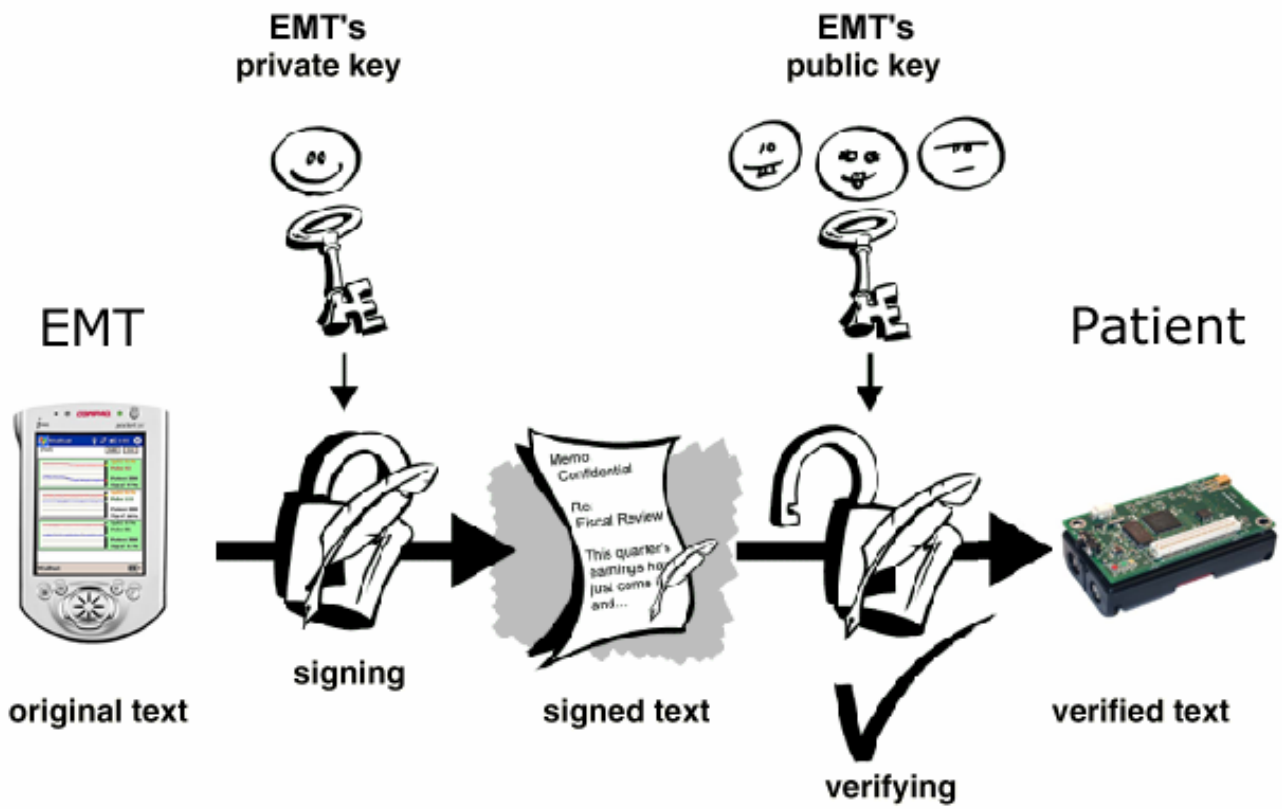


Image adapted from http://www.nuitari.de/crypto.html.

Ensuring Privacy and Security

Research question

How do you design, implement, and evaluate PKI for key distribution in sensor networks?

Ensuring Privacy and Security

Prevailing wisdom

You don’t.

Ensuring Privacy and Security

The real problem: implementation of PKI for sensor networks

- "Public key cryptography is prohibitively expensive for sensor networks in terms of computation and energy consumption."
  - C. Karlof, N. Sastry, and D. Wagner, "TinySec: Link Layer Security for Tiny Devices," `http://www.cs.berkeley.edu/~nks/tinysec/`
- "Many current sensor devices have limited computational power, making public-key cryptographic primitives too expensive in terms of system overhead."
  - 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

Ensuring Privacy and Security

Context for this problem

- The MICA2 Mote
  - Developed at UC Berkeley
  - Fabricated by Crossbow, Inc.
  - Supported by TinyOS, UC Berkeley's open-source operating system for sensor networks
  - Limited Resources
    - 8-bit, 7.3828-MHz ATmega 128L processor
    - 4 KB of primary memory (SRAM)
    - 128 KB of program space (ROM)
    - 512 KB of EEPROM
    - 433-MHz radio: 38.4K baud rate, 29B per-packet payload

Image exerpted from http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0042-05_A_MICA2.pdf.

Ensuring Privacy and Security

Confirming or denying prevailing wisdom

- How to design a PKI for key distribution?
  - Explore choices of underlying primitives
- How to implement PKI for key distribution?
  - Optimize underlying primitives
- How to evaluate PKI for key distribution?
  - Evaluate actual implementations

Our Goal in Three Phases

Is PKI, in fact, viable for key distribution for the MICA2?

1. Is encryption of all traffic realistic with TinyOS's existing cipher?
2. Is key distribution viable with classic solutions?
3. Can we do better? How?

Our Method

Choosing, optimizing, and evaluating primitives

1. Is encryption of all traffic realistic with TinyOS's existing cipher?
   - Evaluation of UC Berkeley's TinySec, TinyOS's existing secret-key cryptographic layer for the MICA2 based on SKIPJACK with 80-bit keys
2. Is key distribution viable with classic solutions?
   - Evaluation of BBN Technologies' implementation of Diffie-Hellman, per NIST's recommendation
     - 1,024-bit keys (*i.e.*, 160-bit exponents and 1,024-bit moduli)
3. Can we do better? How?
   - Evaluation of our own implementations of elliptic curve cryptography (ECC), per NIST's recommendation
     - 163-bit keys

Spoiler Ahead!

PKI is, in fact, viable for key distribution on the MICA2

1. SKIPJACK with 80-bit keys
   - Fast
   - Negligible impact on radio throughput
2. Diffie-Hellman with 1,024-bit keys
   - Relatively slow
   - Key size unappealing
3. ECC with 163-bit keys
   - Promising performance
   - Key size appealing

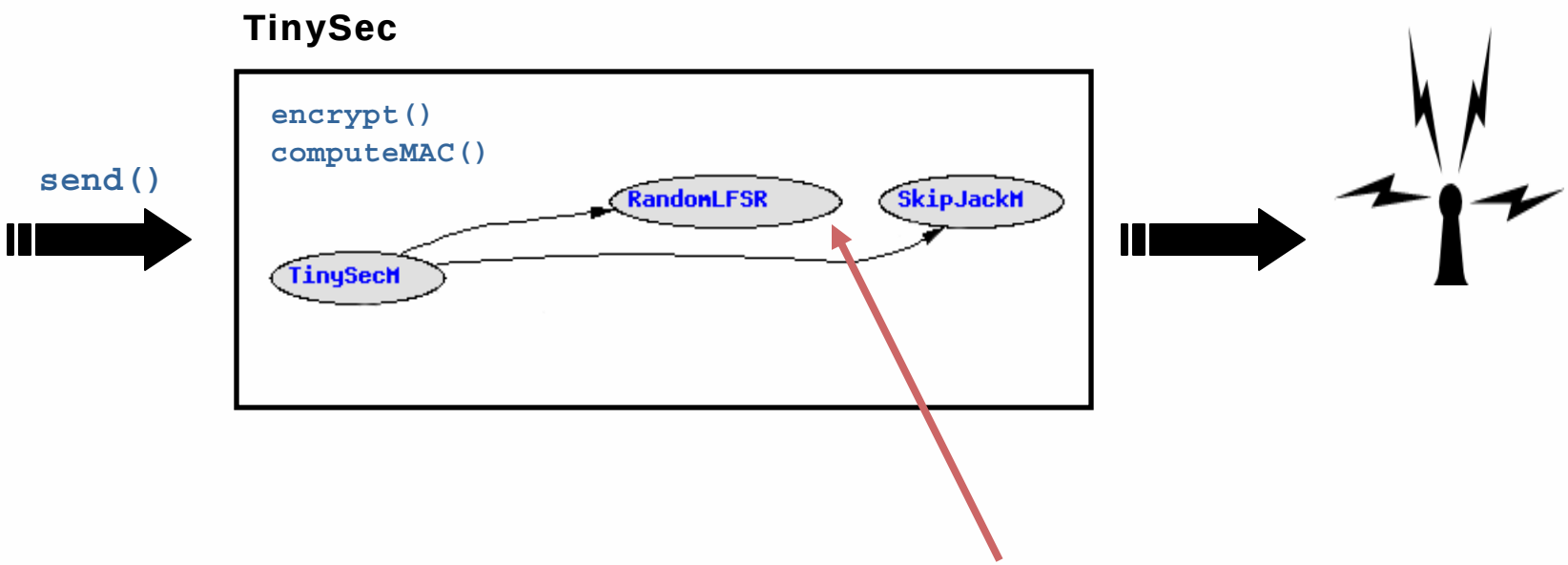
SKIPJACK and the MICA2

Phase 1 of 3

Is encryption of all traffic realistic with TinyOS's existing cipher?

SKIPJACK and the MICA2

TinySec intercepts calls to `send()`



We'll later question our assumption of TinySec's security because of this module, a weak pseudorandom number generator

SKIPJACK and the MICA2

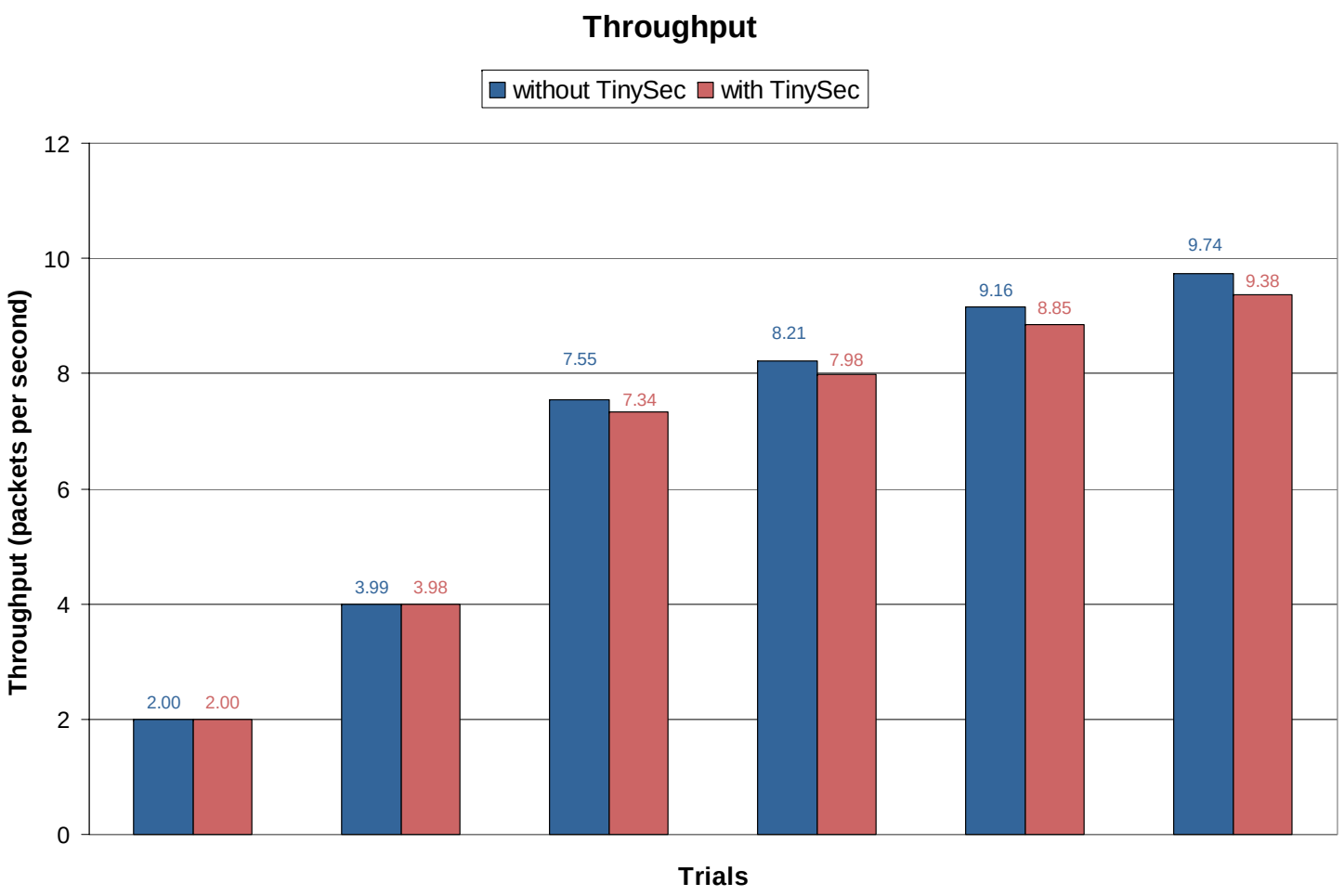
Costs are reasonable

- Costs in time
  - `encrypt()`
    - 2,190 µsec
  - `computeMAC()`
    - 3,049 µsec
- Costs in space
  - 822 B of SRAM
  - 7,076 B of ROM

See IEEE SECON 2004 paper for breakdown of SRAM into `.bss`, `.data`, and stack requirements.

SKIPJACK and the MICA2

Impact on throughput is negligible (0.36 packets per sec)



SKIPJACK and the MICA2

Phase 1 of 3

Encryption of all traffic is realistic with TinyOS's existing cipher: operations are fast, impact on throughput is negligible.

Diffie-Hellman and the MICA2

Phase 2 of 3

Is key distribution viable with classic solutions?

Diffie-Hellman and the MICA2

Why not RSA?

- Timing tests for RSA [on the MICA], public key operation with exponent 3 and varying modulus sizes:

512-bit	3.7 s
768-bit	8.0 s
1024-bit	14.5 s

- Private key timing not available: far too slow.
- Key-exchange: Exponentiation with a 512-bit modulus and 120 bit exponent with 50% one-bits: 315 sec!!

BBN TECHNOLOGIES
A Verizon Company

Results excerpted from http://www.is.bbn.com/projects/lws-nest/bbn_nest_dec_03.ppt.

Diffie-Hellman and the MICA2

Discrete Logarithm Problem (DLP)

Determining x given $g^x \bmod p$ is hard,
where g is a generator and p is a prime.

Modular exponentiation is the primitive underlying
a typical Diffie-Hellman exchange and variants thereof.

Diffie-Hellman and the MICA2

A typical exchange

agree on generator g, prime p

Alice	Bob
choose random A	choose random B

$T_A = g^A \bmod p$ →

← $T_B = g^B \bmod p$

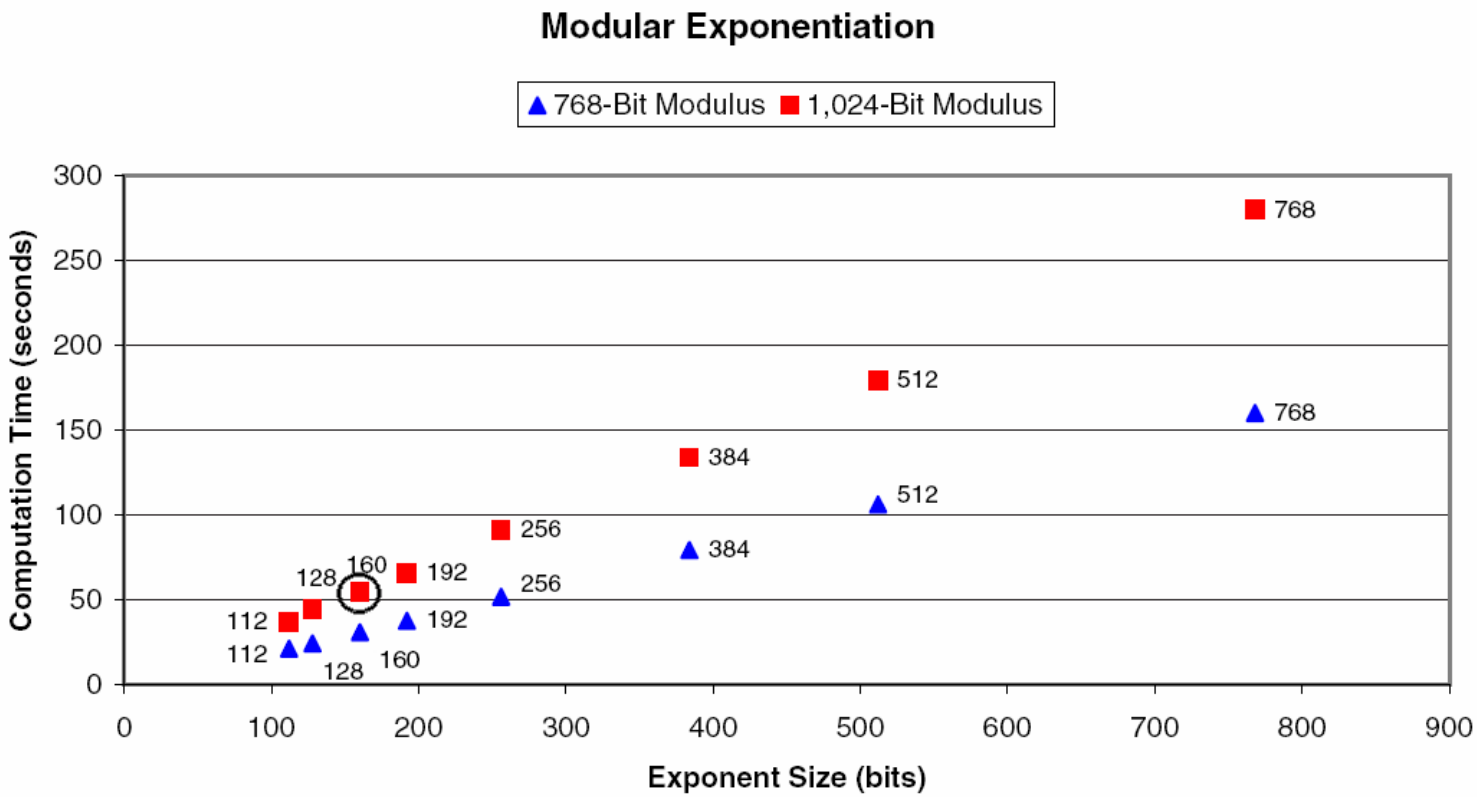
compute T_B^A | compute T_A^B

agree on $g^{AB} \bmod p$

Image adapted from Radia Perlman's Computer Science 243.

Diffie-Hellman and the MICA2

Performance of primitive is relative slow



Time required to compute $2^x \pmod{p}$, where p is prime, on the MICA2, averaged over 100 trials.

Diffie-Hellman and the MICA2

Costs of generating a 1,024-bit public or shared key are significant

- Cost in time
  - 54.1144 sec
- Costs in space
  - 1,250 B of SRAM
  - 11,350 B of ROM
- Cost in energy
  - 1.185 Joules (3.9897×10^8 cycles)

See IEEE SECON 2004 paper for breakdown of SRAM into `.bss`, `.data`, and stack requirements.

Diffie-Hellman and the MICA2

Putting energy costs in perspective

- Up to 2 × 2,850 mAh × 3600 sec/h ÷ (7.3 mA × 54.1144 sec) ≈ 51,945 modular exponentiations are possible with two AA batteries on the MICA2.
  - Energizer No. E91, an AA battery, offers an average capacity of 2,850 mAh.
  - The average current drawn with modular exponentiation in progress was 7.3 mA.
- This bound is generous: the MICA2 effectively dies once voltage drops below 2 V.

Diffie-Hellman and the MICA2

Phase 2 of 3

Key distribution is costly with classic solutions:
relatively slow, key size unappealing.

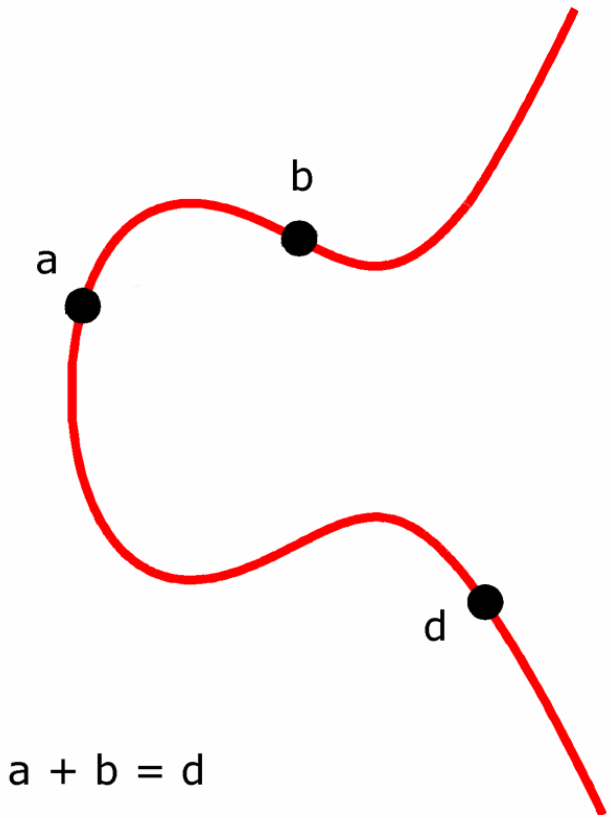
ECC and the MICA2

Phase 3 of 3

Can we do better (than 54.1144 sec and 1,024 bits)? How?

ECC and the MICA2

Elliptic curve cryptography (ECC) offers an alternative primitive



ECC and the MICA2

Elliptic Curve Discrete Logarithm Problem (ECDLP)

Determining k given $k \times G$ is hard,
where G is a publicly known point on a publicly known curve.

Point multiplication is the primitive underlying ECC.

ECC and the MICA2

A typical exchange

curve and point G are publicly known

Alice	Bob
choose random k_A	choose random k_B

$$T_A = k_A * G \quad \longrightarrow$$

$$\longleftarrow \quad T_B = k_B * G$$

compute $k_A * T_B$	compute $k_B * T_A$

agree on $k_A * k_B * G$

ECC and the MICA2

Why ECC?

With ECC, secure distribution of 80-bit TinySec keys is possible using public keys with fewer bits than 1,024:
163 bits are sufficient.

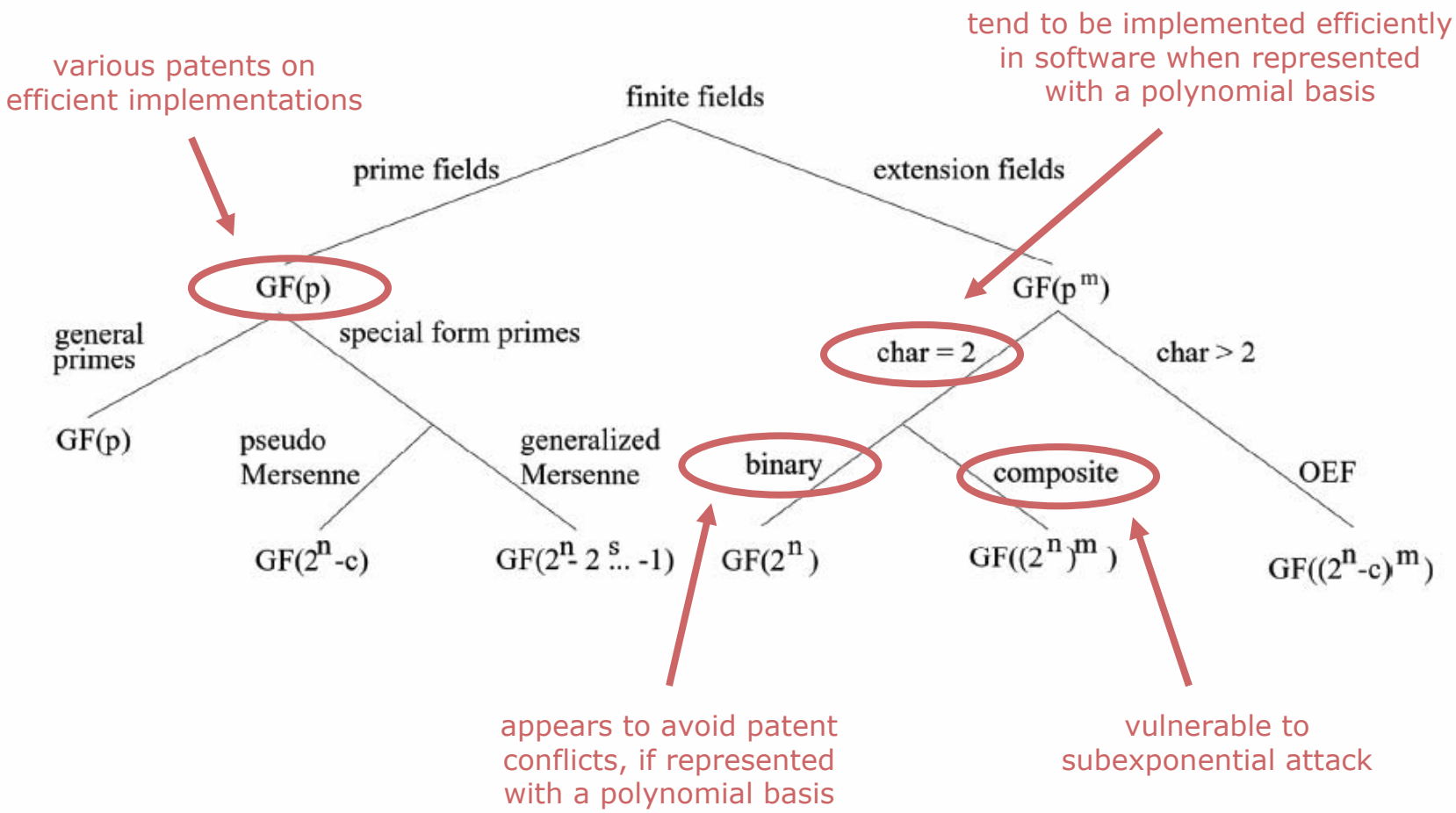
ECC and the MICA2

Secure distribution of 80-bit TinySec keys with 163-bit public keys

- Elliptic curves are believed to offer security computationally equivalent to that of Diffie-Hellman based on DLP with remarkably smaller key sizes
  - Subexponential algorithms exist for DLP
    - 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
    - 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
    - M. Rabin, "Digitalized signatures and public-key functions as intractable as factorization," MIT, Tech. Rep. MIT/LCS/TR-212, 1979
  - No subexponential algorithm is known to exist for ECDLP over certain finite fields
    - P. Gaudry, F. Hess, and N. P. Smart, "Constructive and Destructive Facets of Weil Descent on Elliptic Curves," Department of Computer Science, University of Bristol, Tech. Rep. CSTR-00-016, October 2000

ECC and the MICA2

Galois (*i.e.*, finite) fields proposed for use in PKI



ECC and the MICA2

Reinforcing popular wisdom

Our first implementation of ECC on the MICA2 was a failure.

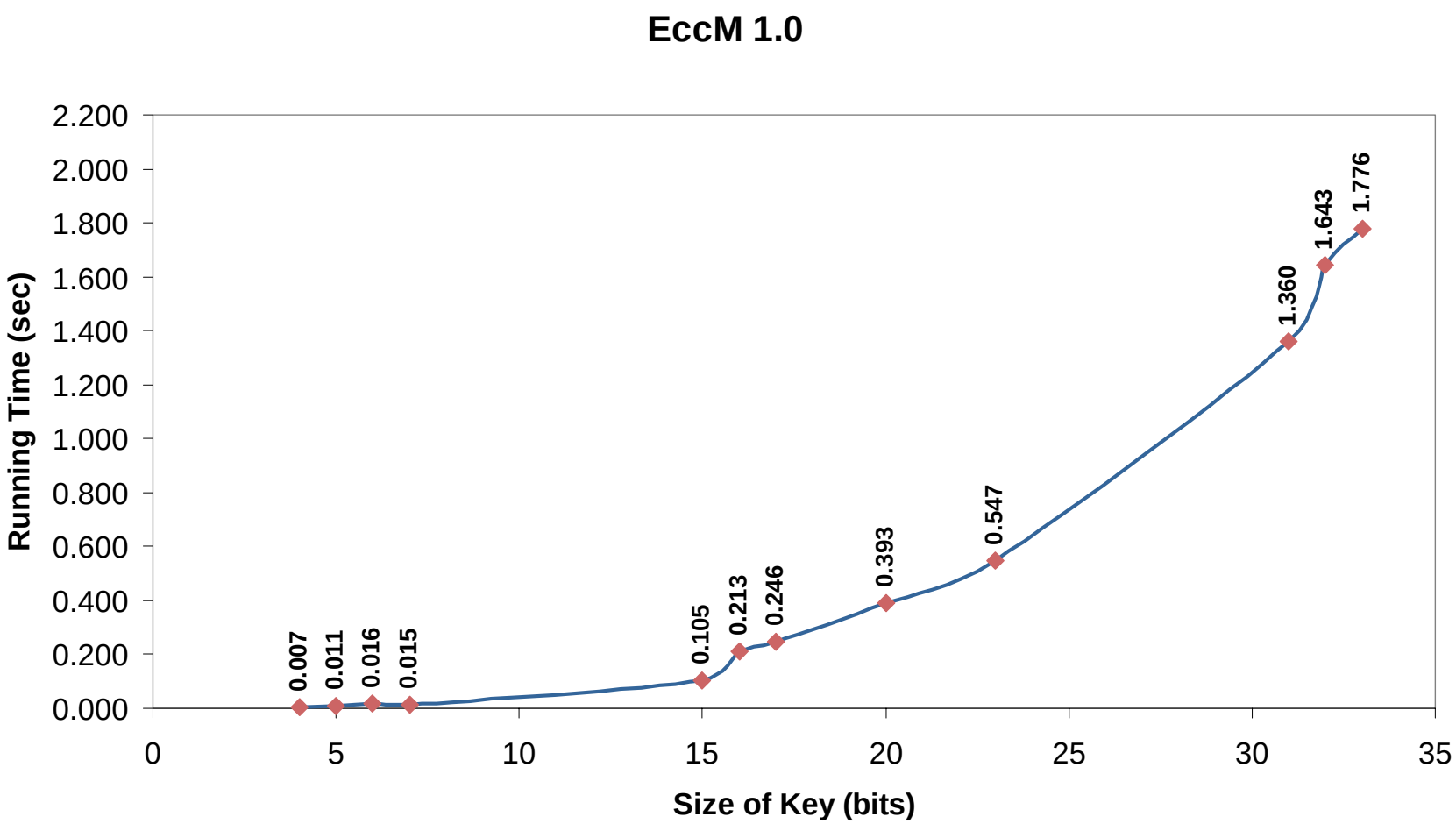
ECC and the MICA2

EccM 1.0: our first implementation

- A TinyOS module based on code ported from Michael Rosing's *Implementing Elliptic Curve Cryptography* that implements point multiplication.
- Operates over $GF(2^p)$, using a polynomial basis, modulo an irreducible polynomial.
- Features:
  - selects a random curve of the form $y^2 + xy \equiv x^3 + ax^2 + b$, where $a = 0$ and $b \in GF(2^p)$;
  - selects from that curve a random point, $G \in GF(2^p) \times GF(2^p)$;
  - selects randomly a private key, $k \in GF(2^p)$; and
  - computes $k \cdot G$, the corresponding public key.

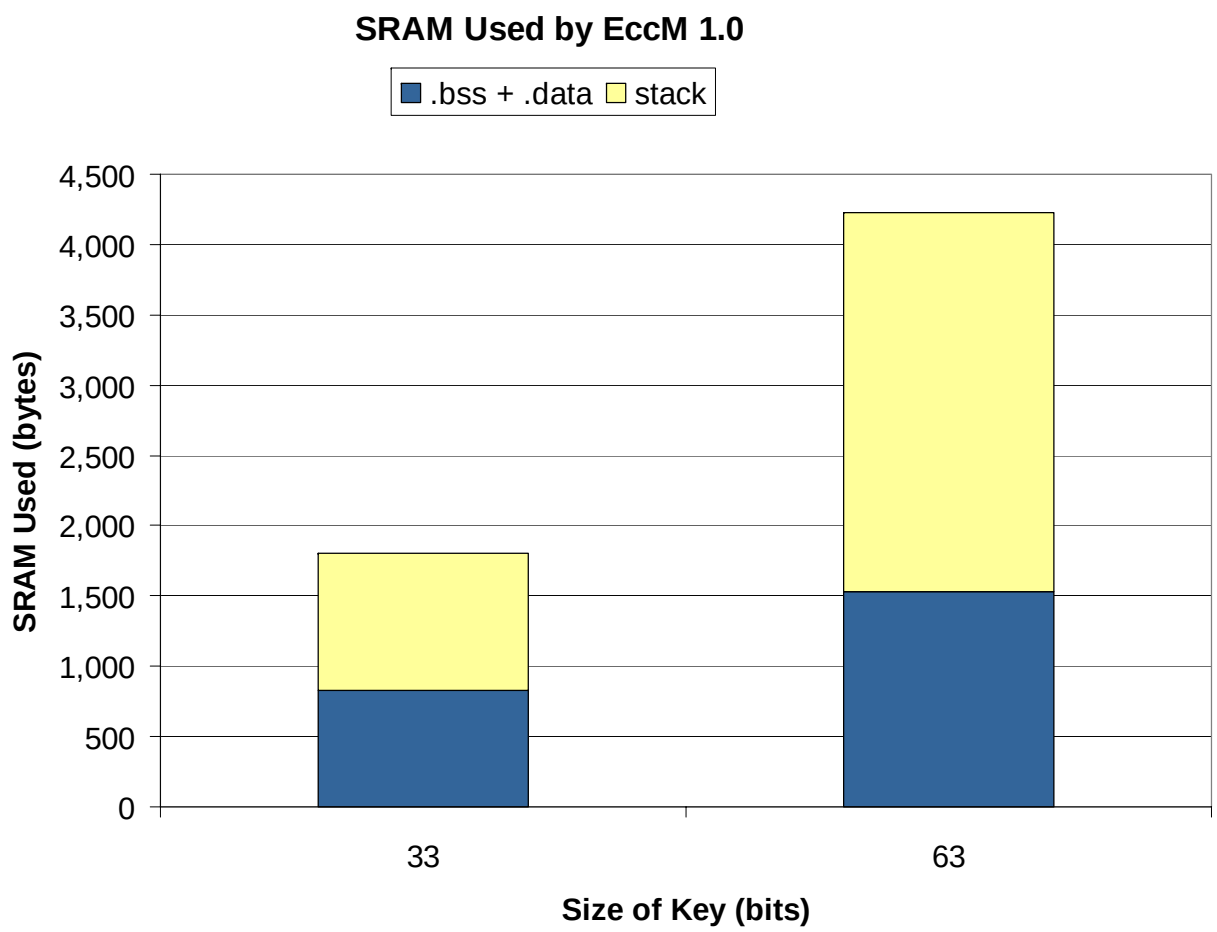
ECC and the MICA2

EccM 1.0: cost in time



ECC and the MICA2

EccM 1.0: costs in space



ECC and the MICA2

EccM 2.0: our second implementation

- A TinyOS module inspired by Dragongate Technologies Limited's Java-based jBorZoi 0.9 that implements actual key exchange.
- Operates over GF(2^{163}), using a polynomial basis, modulo $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$.
- Features:
  - selects a Koblitz curve, $y^2 + xy \equiv x^3 + x^2 + 1$, the number of points on which is 5846006549323611672814741753598448348329118574063;
  - selects a base point, $G = (G_x, G_y)$, where G_x = 4373527398576640063579304354969275615843559206632 and G_y = 3705292482178961271312284701371585420180764402649;
  - selects randomly for a node, Alice, a private key, $k_A \in$ GF(2^p);
  - computes Alice's public key, $T_A = k_A \cdot G$;
  - transmits T_A to a node, Bob, who similarly generates and transmits his own T_B;
  - computes for Alice $k_A \cdot T_B = k_A \cdot k_B \cdot G$ just as Bob computes $k_B \cdot T_A = k_A \cdot k_B \cdot G$, the same shared secret.

ECC and the MICA2

Costs of generating 163-bit public or shared key

- Cost in time
  - 495.92 sec
- Costs in space
  - 901 B of SRAM
  - 43,286 B of ROM
- Cost in energy
  - 12.65 Joules (3.65 x 10^9 cycles)

ECC and the MICA2

Optimizing EccM 2.0

- Elimination of foolish code
  (*e.g.*, recomputing terminal conditions for loops)
- Implementation of published algorithms from current literature
  (*e.g.*, J. López and R. Dahab, "High-Speed Software Multiplication in F_{2^m}," May 2000)
- Resurrected lost art of hand-optimized source
  - manually unrolled loops
  - special-cased common shifts
  - re-ordered loops based on expected lengths of elements

ECC and the MICA2

Optimizing EccM 2.0: methodology

```
       b_xor: 185368
 b_bitlength: 166631
 b_shiftleft: 125634
   b_testbit: 115817
b_shiftleft1: 115655
      b_copy: 82415
       f_add: 78891
  b_compareto: 3632
       b_sub: 2805
     b_clear: 2149
       f_mod: 952
       f_mul: 714
    b_iszero: 641
    p_iszero: 478
   b_isequal: 399
       c_add: 240
       f_inv: 238
    b_setbit: 5
     b_print: 3
      p_copy: 2
```

ECC and the MICA2

Optimizing EccM 2.0: illustrative example

```
/**
 * c = a XOR b.
 */
inline void b_xor(uint8_t * a, uint8_t * b, uint8_t * c)
{
    index_t i;
    for (i = 0; i < NUMWORDS; i++, a++, b++, c++)
        *c = *a ^ *b;
}
```

ECC and the MICA2

Optimizing EccM 2.0: helping the compiler

```
000073f4 <EccM$b_xor>:
    73f4:   cf 93           push    r28
    73f6:   df 93           push    r29
    73f8:   cd b7           in  r28, 0x3d   ; 61
    73fa:   de b7           in  r29, 0x3e   ; 62
    73fc:   28 97           sbiw    r28, 0x08   ; 8
    73fe:   0f b6           in  r0, 0x3f    ; 63
    7400:   f8 94           cli
    7402:   de bf           out 0x3e, r29   ; 62
    7404:   0f be           out 0x3f, r0    ; 63
    7406:   cd bf           out 0x3d, r28   ; 61
    7408:   89 83           std Y+1, r24    ; 0x01
    740a:   9a 83           std Y+2, r25    ; 0x02
    740c:   6b 83           std Y+3, r22    ; 0x03
    740e:   7c 83           std Y+4, r23    ; 0x04
    7410:   4d 83           std Y+5, r20    ; 0x05
    7412:   5e 83           std Y+6, r21    ; 0x06
    7414:   1f 82           std Y+7, r1 ; 0x07
    7416:   18 86           std Y+8, r1 ; 0x08
    7418:   8f 81           ldd r24, Y+7    ; 0x07
    741a:   98 85           ldd r25, Y+8    ; 0x08
    741c:   8a 32           cpi r24, 0x2A   ; 42
    741e:   91 05           cpc r25, r1
    7420:   0c f0           brlt    .+2         ; 0x7424
    7422:   20 c0           rjmp    .+64        ; 0x7464
    7424:   2d 81           ldd r18, Y+5    ; 0x05
    7426:   3e 81           ldd r19, Y+6    ; 0x06
    7428:   e9 81           ldd r30, Y+1    ; 0x01
    742a:   fa 81           ldd r31, Y+2    ; 0x02
    742c:   ab 81           ldd r26, Y+3    ; 0x03
    742e:   bc 81           ldd r27, Y+4    ; 0x04
    7430:   90 81           ld  r25, Z
    7432:   8c 91           ld  r24, X
    7434:   89 27           eor r24, r25
    7436:   f9 01           movw    r30, r18
    7438:   80 83           st  Z, r24
    743a:   8f 81           ldd r24, Y+7    ; 0x07
    743c:   98 85           ldd r25, Y+8    ; 0x08
    743e:   01 96           adiw    r24, 0x01   ; 1
    7440:   8f 83           std Y+7, r24    ; 0x07
    7442:   98 87           std Y+8, r25    ; 0x08
    7444:   89 81           ldd r24, Y+1    ; 0x01
    7446:   9a 81           ldd r25, Y+2    ; 0x02
    7448:   01 96           adiw    r24, 0x01   ; 1
    744a:   89 83           std Y+1, r24    ; 0x01
    744c:   9a 83           std Y+2, r25    ; 0x02
    744e:   8b 81           ldd r24, Y+3    ; 0x03
    7450:   9c 81           ldd r25, Y+4    ; 0x04
    7452:   01 96           adiw    r24, 0x01   ; 1
    7454:   8b 83           std Y+3, r24    ; 0x03
    7456:   9c 83           std Y+4, r25    ; 0x04
    7458:   8d 81           ldd r24, Y+5    ; 0x05
    745a:   9e 81           ldd r25, Y+6    ; 0x06
    745c:   01 96           adiw    r24, 0x01   ; 1
    745e:   8d 83           std Y+5, r24    ; 0x05
    7460:   9e 83           std Y+6, r25    ; 0x06
    7462:   da cf           rjmp    .-76        ; 0x7418
    7464:   28 96           adiw    r28, 0x08   ; 8
    7466:   0f b6           in  r0, 0x3f    ; 63
    7468:   f8 94           cli
    746a:   de bf           out 0x3e, r29   ; 62
    746c:   0f be           out 0x3f, r0    ; 63
    746e:   cd bf           out 0x3d, r28   ; 61
    7470:   df 91           pop r29
    7472:   cf 91           pop r28
    7474:   08 95           ret
```

ECC and the MICA2

Optimizing EccM 2.0: helping the compiler

```
/**
 * c = a XOR b.
 */
inline void b_xor(uint8_t * a, uint8_t * b, uint8_t * c)
{
    index_t i;
    for (i = 0; i < NUMWORDS; i++, a++, b++, c++)
        *c = *a ^ *b;
}
```

⇓

```
/**
 * c = a XOR b.
 */
inline void b_xor(uint8_t * a, uint8_t * b, uint8_t * c)
{
    index_t i;
    for (i = 0; i < NUMWORDS; i += 2, a += 2, b += 2, c += 2)
        *((uint16_t *) c) = *((uint16_t *) a) ^ *((uint16_t *) b);
}
```

ECC and the MICA2

Optimizing EccM 2.0: results

```
000073f4 <EccM$b_xor>:
    73f4:   cf 93           push    r28
    73f6:   df 93           push    r29
    73f8:   cd b7           in  r28, 0x3d   ; 61
    73fa:   de b7           in  r29, 0x3e   ; 62
    73fc:   28 97           sbiw    r28, 0x08   ; 8
    73fe:   0f b6           in  r0, 0x3f    ; 63
    7400:   f8 94           cli
    7402:   de bf           out 0x3e, r29   ; 62
    7404:   0f be           out 0x3f, r0    ; 63
    7406:   cd bf           out 0x3d, r28   ; 61
    7408:   89 83           std Y+1, r24    ; 0x01
    740a:   9a 83           std Y+2, r25    ; 0x02
    740c:   6b 83           std Y+3, r22    ; 0x03
    740e:   7c 83           std Y+4, r23    ; 0x04
    7410:   4d 83           std Y+5, r20    ; 0x05
    7412:   5e 83           std Y+6, r21    ; 0x06
    7414:   1f 82           std Y+7, r1 ; 0x07
    7416:   18 86           std Y+8, r1 ; 0x08
    7418:   8f 81           ldd r24, Y+7    ; 0x07
    741a:   98 85           ldd r25, Y+8    ; 0x08
    741c:   8a 32           cpi r24, 0x2A   ; 42
    741e:   91 05           cpc r25, r1
    7420:   0c f0           brlt    .+2         ; 0x7424
    7422:   20 c0           rjmp    .+64        ; 0x7464
    7424:   2d 81           ldd r18, Y+5    ; 0x05
    7426:   3e 81           ldd r19, Y+6    ; 0x06
    7428:   e9 81           ldd r30, Y+1    ; 0x01
    742a:   fa 81           ldd r31, Y+2    ; 0x02
    742c:   ab 81           ldd r26, Y+3    ; 0x03
    742e:   bc 81           ldd r27, Y+4    ; 0x04
    7430:   90 81           ld  r25, Z
    7432:   8c 91           ld  r24, X
    7434:   89 27           eor r24, r25
    7436:   f9 01           movw    r30, r18
    7438:   80 83           st  Z, r24
    743a:   8f 81           ldd r24, Y+7    ; 0x07
    743c:   98 85           ldd r25, Y+8    ; 0x08
    743e:   01 96           adiw    r24, 0x01   ; 1
    7440:   8f 83           std Y+7, r24    ; 0x07
    7442:   98 87           std Y+8, r25    ; 0x08
    7444:   89 81           ldd r24, Y+1    ; 0x01
    7446:   9a 81           ldd r25, Y+2    ; 0x02
    7448:   01 96           adiw    r24, 0x01   ; 1
    744a:   89 83           std Y+1, r24    ; 0x01
    744c:   9a 83           std Y+2, r25    ; 0x02
    744e:   8b 81           ldd r24, Y+3    ; 0x03
    7450:   9c 81           ldd r25, Y+4    ; 0x04
    7452:   01 96           adiw    r24, 0x01   ; 1
    7454:   8b 83           std Y+3, r24    ; 0x03
    7456:   9c 83           std Y+4, r25    ; 0x04
    7458:   8d 81           ldd r24, Y+5    ; 0x05
    745a:   9e 81           ldd r25, Y+6    ; 0x06
    745c:   01 96           adiw    r24, 0x01   ; 1
    745e:   8d 83           std Y+5, r24    ; 0x05
    7460:   9e 83           std Y+6, r25    ; 0x06
    7462:   da cf           rjmp    .-76        ; 0x7418
    7464:   28 96           adiw    r28, 0x08   ; 8
    7466:   0f b6           in  r0, 0x3f    ; 63
    7468:   f8 94           cli
    746a:   de bf           out 0x3e, r29   ; 62
    746c:   0f be           out 0x3f, r0    ; 63
    746e:   cd bf           out 0x3d, r28   ; 61
    7470:   df 91           pop r29
    7472:   cf 91           pop r28
    7474:   08 95           ret
```

saves us 212 cycles per call to `b_xor();` costs us ROM

```
000073f4 <EccM$b_xor>:
    73f4:   cf 93           push    r28
    73f6:   df 93           push    r29
    73f8:   cd b7           in  r28, 0x3d   ; 61
    73fa:   de b7           in  r29, 0x3e   ; 62
    73fc:   28 97           sbiw    r28, 0x08   ; 8
    73fe:   0f b6           in  r0, 0x3f    ; 63
    7400:   f8 94           cli
    7402:   de bf           out 0x3e, r29   ; 62
    7404:   0f be           out 0x3f, r0    ; 63
    7406:   cd bf           out 0x3d, r28   ; 61
    7408:   89 83           std Y+1, r24    ; 0x01
    740a:   9a 83           std Y+2, r25    ; 0x02
    740c:   6b 83           std Y+3, r22    ; 0x03
    740e:   7c 83           std Y+4, r23    ; 0x04
    7410:   4d 83           std Y+5, r20    ; 0x05
    7412:   5e 83           std Y+6, r21    ; 0x06
    7414:   1f 82           std Y+7, r1 ; 0x07
    7416:   18 86           std Y+8, r1 ; 0x08
    7418:   8f 81           ldd r24, Y+7    ; 0x07
    741a:   98 85           ldd r25, Y+8    ; 0x08
    741c:   8a 32           cpi r24, 0x2A   ; 42
    741e:   91 05           cpc r25, r1
    7420:   0c f0           brlt    .+2         ; 0x7424
    7422:   25 c0           rjmp    .+74        ; 0x746e
    7424:   4d 81           ldd r20, Y+5    ; 0x05
    7426:   5e 81           ldd r21, Y+6    ; 0x06
    7428:   e9 81           ldd r30, Y+1    ; 0x01
    742a:   fa 81           ldd r31, Y+2    ; 0x02
    742c:   ab 81           ldd r26, Y+3    ; 0x03
    742e:   bc 81           ldd r27, Y+4    ; 0x04
    7430:   20 81           ld  r18, Z
    7432:   31 81           ldd r19, Z+1    ; 0x01
    7434:   8d 91           ld  r24, X+
    7436:   9c 91           ld  r25, X
    7438:   11 97           sbiw    r26, 0x01   ; 1
    743a:   82 27           eor r24, r18
    743c:   93 27           eor r25, r19
    743e:   fa 01           movw    r30, r20
    7440:   80 83           st  Z, r24
    7442:   91 83           std Z+1, r25    ; 0x01
    7444:   8f 81           ldd r24, Y+7    ; 0x07
    7446:   98 85           ldd r25, Y+8    ; 0x08
    7448:   02 96           adiw    r24, 0x02   ; 2
    744a:   8f 83           std Y+7, r24    ; 0x07
    744c:   98 87           std Y+8, r25    ; 0x08
    744e:   89 81           ldd r24, Y+1    ; 0x01
    7450:   9a 81           ldd r25, Y+2    ; 0x02
    7452:   02 96           adiw    r24, 0x02   ; 2
    7454:   89 83           std Y+1, r24    ; 0x01
    7456:   9a 83           std Y+2, r25    ; 0x02
    7458:   8b 81           ldd r24, Y+3    ; 0x03
    745a:   9c 81           ldd r25, Y+4    ; 0x04
    745c:   02 96           adiw    r24, 0x02   ; 2
    745e:   8b 83           std Y+3, r24    ; 0x03
    7460:   9c 83           std Y+4, r25    ; 0x04
    7462:   8d 81           ldd r24, Y+5    ; 0x05
    7464:   9e 81           ldd r25, Y+6    ; 0x06
    7466:   02 96           adiw    r24, 0x02   ; 2
    7468:   8d 83           std Y+5, r24    ; 0x05
    746a:   9e 83           std Y+6, r25    ; 0x06
    746c:   d5 cf           rjmp    .-86        ; 0x7418
    746e:   28 96           adiw    r28, 0x08   ; 8
    7470:   0f b6           in  r0, 0x3f    ; 63
    7472:   f8 94           cli
    7474:   de bf           out 0x3e, r29   ; 62
    7476:   0f be           out 0x3f, r0    ; 63
    7478:   cd bf           out 0x3d, r28   ; 61
    747a:   df 91           pop r29
    747c:   cf 91           pop r28
    747e:   08 95           ret
```

ECC and the MICA2

EccM 2.0 ultimately offered smaller keys *and* better performance

- Cost in time
  - 34.161 sec [down from 495.92 sec]
- Costs in space
  - 1,140 B of SRAM [up from 901 B]
  - 34,342 B of ROM [down from 43,286 B]
- Cost in energy
  - 0.816 Joules (2.512×10^8 cycles)
    [down from 12.65 Joules (3.65×10^9 cycles)]
    - $2.512 \times 10^8 \div 7.3828$ MHz $\times$ 8 mA $\times$ 3 V

See IEEE SECON 2004 paper for breakdown of SRAM into `.bss`, `.data`, and stack requirements.

ECC and the MICA2

Putting energy costs in perspective

- Up to 2 × 2,850 mAh × 3600 sec/h ÷ (8.0 mA × 34.161 sec) ≈ 75,086 point multiplications are possible [up from 51,945 modular exponentiations] with two AA batteries on the MICA2.
  - Energizer No. E91, an AA battery, offers an average capacity of 2,850 mAh.
  - The average current drawn with point multiplication in progress was 8.0 mA.
- This bound is, again, generous: the MICA2 effectively dies once voltage drops below 2 V.

ECC and the MICA2

Phase 3 of 3

We *can* do better:
34.161 sec for generation of 163-bit public or shared keys
versus
54.1144 sec for generation of 1,024-bit public or shared keys.

Related Work

Current literature

- C. Karlof, N. Sastry, and D. Wagner, "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks," Second ACM Conference on Embedded Networked Sensor Systems, November 2004
- R. Watro, D. Kong, S. Cuti, C. Gardiner, C. Lynn, P. Kruus, "TinyPK: Securing Sensor Networks with Public Key Technology," Second ACM Workshop on Security of Ad Hoc and Sensor Networks , October 2004
- N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs," Sixth International Workshop on Cryptographic Hardware and Embedded Systems, August 2004
  - 0.81 sec for 160-bit point multiplication over GF(p)

Caveats

Open problems remain

- TinyOS's `RandomLFSR` module is seeded only with a mote's ID number: it's a poor pseudo-random number generator
  - TinySec is not as secure as it might be as a result
- PKI primitives are viable, but implementation details for protocols remain
  - What features do mass casualty incidents demand?
- Chains of trust need to be explored
  - Is a certificate authority needed for authentication and revocation?

Conclusion

PKI is, in fact, viable for key distribution on the MICA2

1. Is encryption of all traffic realistic with TinyOS's existing cipher?
   - SKIPJACK with 80-bit keys
     - 0.36 packets per second of overhead
2. Is key distribution viable with classic solutions?
   - Diffie-Hellman with 1,024-bit keys
     - 54.1144 sec for key generation
3. Can we do better? How?
   - ECC with 163-bit keys
     - 34.390 sec for key generation

Future Work

Toward the dissertation

"There may be a virus loose on the internet."
Andy Sudduth of Harvard, 34 minutes after midnight, Nov. 3, 1988

Image and quote from http://www.software.com.pl/newarchive/misc/Worm/darbyt/pages/what.html.

From Sneakernet to Internet

Malware's threats to availability, integrity, and security

- Viruses
  - self-replicating software that can spread from one system to another only via some host
- Worms
  - self-propagating software that can spread from one system to another independent of any host
- Spyware
  - software that monitors a user's activity and reports that user's activity to some server

Size of the Problem

Omnipresent Windows particularly vulnerable

- Over 68,000 viruses and worms for Windows [Symantec Corporation]
- 1,200 forms of spyware for Windows [Safer Networking Limited]

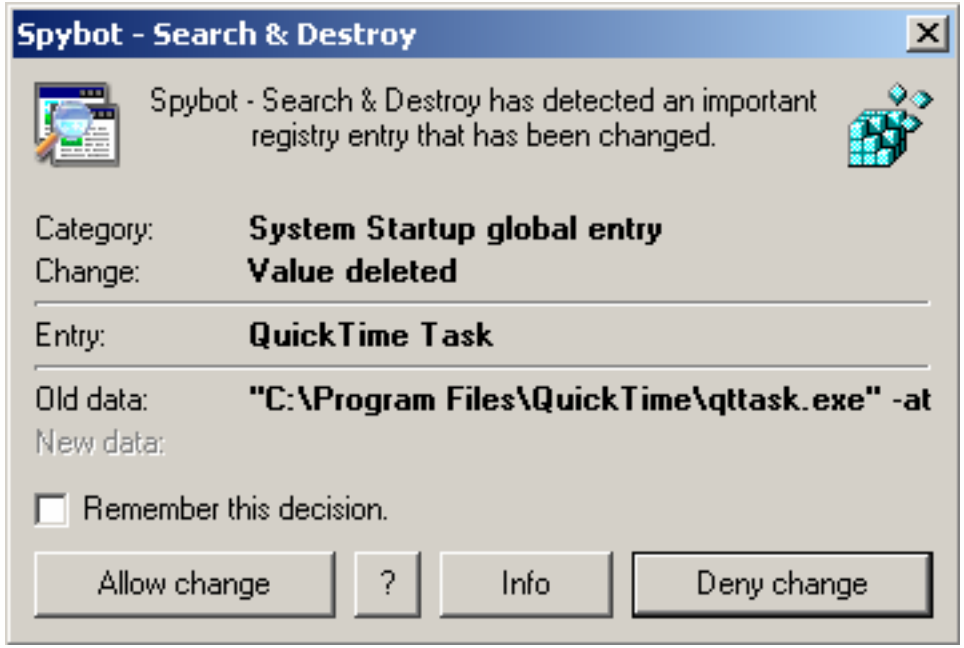
Classic Solutions

Signature-based techniques are inadequate

1. Systems are safe from only those attacks that other systems have already suffered
2. Systems vulnerable for as long as it takes for researchers to define and distribute new signatures
3. Fastest of worms do not allow sufficient time for human intervention
4. All too easily defeated by metamorphic or polymorphic malware

Classic Solutions

Behavior-based techniques suffer from false positives



Thesis Proposal

Challenging classic solutions

Can we do better? How?

Distributed Runtime Defense

Contribution of thesis

Combat inevitable risks of false positives through correlation of seemingly anomalous behavior by Win32 systems in peer-to-peer networks.

Plan of Attack

Phase 1: research question

Does straightforward application of extant techniques for anomaly detection on Linux and UNIX enable detection of viruses, worms, and spyware on Win32?

Plan of Attack

Phase 1: methodology

- Assess applicability of extant techniques to Win32
  - Port from Linux 2.4 to Windows XP the ideas embodied in UNM's pH, a kernel extension which looks to yet-unseen patterns of system calls for definition of anomalous behavior, exponentially slowing execution of all such patterns
  - Hook the NT services provided in kernel space by NTOSKRNL.EXE
  - Vary pH's rules as well as alternative definitions of behavior for Win32 systems

Plan of Attack

Phase 2: research question

Can we leverage ideas from the peer-to-peer community and distribute the problem of detection across hosts vulnerable to the same attacks?

Plan of Attack

Phase 2: methodology

- Implement peer-to-peer cooperation; determine to what extent and with what speed peers can extract value from correlated events
  - Unprecedented behavior is unlikely to be exhibited by multiple peers within a narrow slice of time, unless triggered by some external source
  - We expect a benefit of this phase's enhancements will be reduction of false positives, vis-à-vis phase 1's implementation
  - An additional benefit should be novel defense against "flash" worms, whose rate of propagation can overwhelm any human-led effort to craft custom defenses

Plan of Attack

Phase 3: research question

How best to respond to anomalies once detected?

Summary

Phases 1 through 3

1. Establish a baseline for Win32
2. Attack false positives: extract value from correlated events
3. Distribute defense

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

Qualifying Examination

David J. Malan
Division of Engineering and Applied Sciences
Harvard University

16 November 2004