International Journal of Innovative Technology and Exploring Engineering (IJITEE)
ISSN: 2278-3075 (Online), Volume-2 Issue-4, March 2013

Public Key Encryption Algorithms for Wireless Sensor Networks In tiny OS

Chandni Vaghasia, Kirti Bathwar

Abstract*—generally, when people consider wireless devices they think of items such as cell phones, personal digital assistants, or laptops. These items are costly, target specialized applications, and rely on the pre-deployment of extensive infrastructure support. In contrast, wireless sensor networks use small, low-cost embedded devices for a wide range of applications and do not rely on any pre-existing infrastructure. The emerging field of wireless sensor networks (WSN) combines sensing, computation, and communication into a single tiny device called sensor nodes or motes. Through advanced mesh networking protocols, these devices form a sea of connectivity that extends the reach of cyberspace out into the physical world. here some algorithms are implemented and result is analyzed on different platforms like PC MICA, Mica 2, Mica2dot and analyze which algorithm is best for which platform.

Keywords*— Cryptography, Public Key Encryption, Sensor nodes, *Wireless Sensor Networks.

I. INTRODUCTION

WSNs have an endless array of potential applications but the most important issue is to provide a high level security in transmission of data between very sensor node and base station. Security issue become very important when the data are highly sensitive and highly confidential. hence the private key encryption is widely used for this purpose in most of the network. Although it is return the feasible solution public key encryption has large number of advantage over it. one of its biggest advantage is that public-key based solutions provide a higher level of system security, since nodes would not be equipped with private keys, which would limit the advantage gained by an attacker compromising some of the nodes. so we are inspire to implement the public key encryption algorithm for the wireless sensor networks which have not been implemented till now.

II. BAKEGROUND

A wireless sensor network (WSN) is a wireless network consisting of spatially distributed autonomous devices using sensors to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations. The development of wireless sensor networks was originally motivated by military applications such as battlefield surveillance. However, wireless sensor networks are now used in many civilian applications too [3]. A WSN usually consists of tens to thousands of such nodes that communicate through wireless channels for information sharing and cooperative processing. These are called wireless sensor node. Typically, a wireless sensor node (or simply sensor node) consists of sensing, computing, communication and power components. These components are integrated on a single or multiple boards, and packaged in a few cubic inches. The concept of wireless sensor networks is based on a simple equation:

Sensing + CPU + Signal (Radio) = Thousands of potential applications.

In addition to one or more sensors, each node in a sensor network is typically equipped with a radio transceiver or other wireless communications device, a small microcontroller, and an energy source, usually a battery. The cost of sensor nodes is similarly variable, ranging from hundreds of dollars to a few cents, depending on the size of the sensor network and the complexity required of individual sensor nodes. Size and cost constraints on sensor nodes result in corresponding constraints on resources such as energy, memory, computational speed and bandwidth.

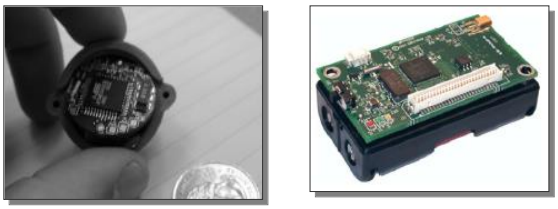


Fig 1: Typical sensor nodes also called Motes

The major elements of WSNs are the **sensor nodes** and the **base stations[1]**. In fact, they can be abstracted as the "sensing cells" and the "brain" of the network, respectively. Usually, sensor nodes are deployed in a designated area by an authority and then, automatically form a network through wireless communications. Sensor nodes keep monitoring the network area after being deployed. After an event of interest occurs, one of the surrounding sensor nodes can detect it, generate a report, and transmit the report to a Base Station (BS) through multi hop wireless links. The BS can process the report and then forward it through either high-quality wireless or wired links to the external world for further processing. Public-key based solutions provide a higher level of system security, since nodes would not be equipped with private keys, which would limit the advantage gained by an attacker compromising some of the nodes. It is therefore important to consider the public-key encryption schemes. There are many public key encryption algorithm named Okamoto Uchiyama Algorithm (OU), Domingo-Ferrer scheme (DF), Castelluccia-MyKletun-Tsudik Scheme (CAMyTs), CAMyTs + Domingo-Ferrer (CaMyTs/DF).

Manuscript published on 30 March 2013.
***Correspondence Author(s)**
Chandni Vaghasia, Computer Engineering Department, CHARUSAT University, Changa,india
Kirti Bathwar, Computer Engineering Department, Bhagawan Mahavir Engineering collage, Surat, India.

III. PUBLIC KEY ENCRYPTION ALGORITHMS

A. *Okamoto-Uchiyama (OU) Implementation*

Okamoto and Uchiyama proposed a new public-key cryptosystem as secure as factoring and based on the ability of computing discrete logarithms in a particular subgroup[4],[5]. This subsection shows how to construct our cryptosystem based on the logarithmic function. Our cryptosystem, based on the exponentiation root n, is constructed as follows:

Their scheme is characterized by probabilistic encryption, additive homomorphic properties, and relating the computational complexity of the encryption function to the size of the plaintext. We now describe their cryptosystem:

Let p and q be random k-bit primes and set $n = p^2q$. Next, randomly choose a $g \in_R Zn$ such that element $g_p = g^{p-1} \pmod{p^2}$ has order p. Finally, set $h = g^n \pmod{n}$.

Okamoto-Uchiyama (OU) [4]	
Public Key	$n = p^2q, g, h$
Private Key	(p, q)
Encryption	plaintext $m \in 2^k$, $r \in_R \mathbb{Z}_n$, ciphertext $c = g^m h^r \pmod{n}$
Decryption	$c' = c^{p-1} \pmod{p^2}$ compute $m = L(c')L(g_p)^{-1} \pmod{p}$
Note that $c^{p-1} \pmod{p^2} = g^{m(p-1)}g^{nr(p-1)} = g_p^m \pmod{p^2}$	

A. *Implimentation*

In this section, we present an implementation of the OU algorithm by mathematical equations.

1) We have to choose two keys p and q as shown in algorithm which are the prime numbers and generates public key n.

Suppose we have taken **p=23** and **q=29**

2) **Encryption:**

a) Calculate the public key n which is given p^2q.

So, n=(23)*(23)*(29)=15341

b) Now as shown in algorithm g is a value generated randomly such that $g \in_R Zn$ such that element $g_p = g^{p-1} \pmod{p^2}$ has order p.For generating it we have implemented primitive root generator code.

c) If we take g=2 which is the least primitive root of p.

$g_p = (2)^{22} \pmod{529} = 392$

d) Calculate the value of *h* as $h = g^n \pmod{n}$

$h = (2)^{15341} \pmod{15341}$

e) We have to enter the plain text m for encryption purpose.and choose r such as r > m.

If m=13 and we have choose r=15 which is greter than m.

f) So calculate the cipher text as $c = g^m * h^r \pmod{n}$.

Ciphertext=$(2)^{13}*(2911)^{15} \pmod{15341}$ = **2737**

3) Decryption:

a) Calculate the $c' = c^{p-1} \pmod{p^2}$.

$C' = (2737)^{22} \pmod{529} = 323$

b) for computing the plain text m from cipher text compute $L(x) = (x - 1)/p$

$L(c') = (c' - 1)/p = (323 - 1)/23 = 14$

$L(g_p) = (g_p - 1)/p = (392 - 1)/23 = 17$

c) for finding the plaintext we have to find out $(L(g_p))^{-1}$ means inverse of $(L(g_p))$.for that purpose we have use the **inverse in galois field** method as shown below.

1) In this method first put the 1 and 0 in respective A1 and A2.and 0 and 1 in the B1 and B2 by default. in the first cell of the A3 put the value of P.

Then recursively find out B1 = A1 – QB1 and

B2 = A2 – QB2.

copy the value of B1 and B2 to the next row in the A1 and A2 respectively.

B3(next row) = A3(next row) % B3(previous row)

2) Continuous above step until the value of the B3 become At last check the value of the B2.there are two possibilities for the result.

a) If value is negative then add it to the P with considering the negative sign.

$(L(g_p)^{-1} = (B2) + P$

b) If value is positive then nothing to do and B3 itself is the answer as a result of inverse $(L(g_p))$.

$(L(g_p)^{-1} = B2$

Q	A1	A2	P A3	B1	B2	Ans B3
	1	**0**	23	**0**	**1**	17
1	0	1	17	1	-1	6
2	1	-1	6	-2	3	5
1	-2	3	5	3	**-4**	1

Table 1: Inverce In Glosic Field Method

For values of P and $L(g_p)$,the above table is generated and the value of B2 after all iteration comes (-4) which is negative.

so $(L(g_p)^{-1} = (B2) + P = -4 + 23 = 19$

d) put all the values in the $m = L(C') * (L(g_p)^{-1} \pmod{p}$.

$m = 14 * 19 \pmod{23}$,**m =13**

b. *Results*

After applying the Okamoto Uchiyama algorithm on the plain block and transmitting over the wireless sensor network we get the same plain block at the receiver side with higher security.

c. *Flow Graph*

Fig 2: Flow Graph of OU

d. *Code Size on different platforms*

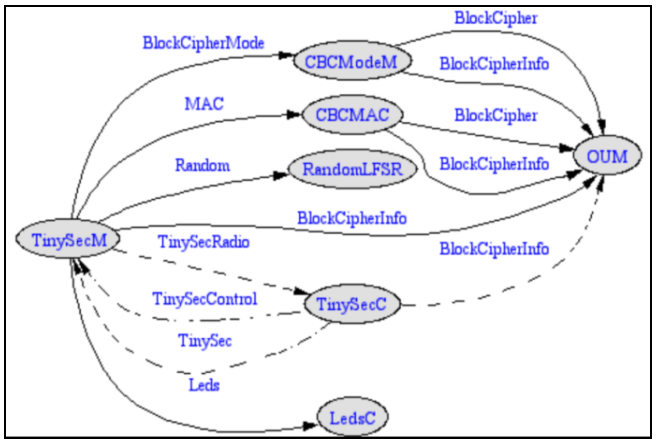
OUM		
Platforms	ROM	RAM
Pc	80896	1372880
Mica	21626	849
Mica2	21126	894
Mica2dot	20954	894

Table 2: OU Code Size on Different Platforms

Published By:
Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP)

B. Domingo-Ferrer (DF) Algorithm

Domingo-Ferrer introduced a symmetric PH scheme (DF) that has been proposed as efficient PH cryptographic system for WSNs.



The PH is probabilistic, which means that the encryption transformation involves some randomness that chooses the cipher text corresponding to a given cleartext from a set of possible cipher texts[8].

Domingo-Ferrer (DF) algorithm [11]

Parameter: *public key: integer* $d \geq 2$, *large integer* M
secret key: $k = (r, g)$
small g *that divides* M; r *so that* r^{-1} *exists in* $\mathbb{Z}_M$

Encryption: split m into d parts $m_1 \ldots m_d$ that
$\sum_{i=1}^{d}(m_i) \bmod g = m$
$C = [c_1, \ldots, c_d] = [m_1 r \bmod M, m_2 r^2 \bmod M, \ldots, m_d r^d \bmod M]$

Decryption: $m = (c_1 r^{-1} + c_2 r^{-2} + \ldots + c_d r^{-d}) \bmod g$

Aggregation: Scalar addition modulo M
$C12 = C1 + C2 = [(c1_1 + c2_1) \bmod M, \ldots, (c1_d + c2_d) \bmod M]$

The set of clear text is **Zg**, and the set of cipher text is $(\mathbf{Z_M})^{\mathbf{d}}$. DF has both the additive and the multiplicative PH properties[12]. For the cipher text multiplication, all terms are cross-multiplied in **Zg**, with the d1-degree term by a d2-degree term yielding a (d1 + d2)-degree term. Terms having the same degree are added up. DF is a symmetric algorithm that requires the same secret key for encryption and decryption. The aggregation is performed with a key that can be publicly known, i.e., the aggregator nodes do not need to be able to decrypt the encrypted messages[8]. However, it is required that the same secret key is applied on every node in the network that needs to encrypt data. The message size is d . n bit. For very secure parameter combinations (d > 100), the messages become very big.

a. Domingo-Ferrer (DF) Algorithm Implementation

In this section, we present an implementation of the DF algorithm by mathematical equations.

1) Key generation:

a)We have to choose **r** and **g** such that **g** can divide **M** and $\mathbf{r^{-1}}$ must be in **Zn,** Which generate secret key **k** as pair of (r , g). **M** should be large number.

Suppose, we have choose r = 2 and g = 10000000 which can divide M=20000000.

Take Secret key K=3.

2) Encryption:

a)Now as shown in the algorithm we have to split the plain text(***m***) in to **n** number of part such that

So, if Plain Text is (***m***) = **52** then split it in to parts as shown below.

$\sum_{i=1}^{d}(m_i) \bmod g = m$

52 is Even no so split it in to Two parts as :

m_1=2,m_2=50

b) Calculate the Cipher text as

$$C = [c_1, \ldots, c_d] = [m_1 r \bmod M, m_2 r^2 \bmod M, \ldots, m_d r^d \bmod M]$$

Cipher Text **c** = (c_1) = *m* 1* r * mod M =
2*2 mod 20000000 = 4
(c_2) = *m* 2* r * mod M = 50*2 mod 20000000 = 100

3) Decryption:

a) For computing the plain text ***m*** from cipher text compute

$$m = (c_1 r^{-1} + c_2 r^{-2} + \ldots + c_d r^{-d}) \bmod g$$

(m_1) = c1*r^{-1}=4 *2^{-1} =2
(m_2) = c2*r^{-1}=100*2^{-1} =50

m = (m_1+ m_2) mod g = (50+2) mod 10000000 = **52**

b. Flow Graph

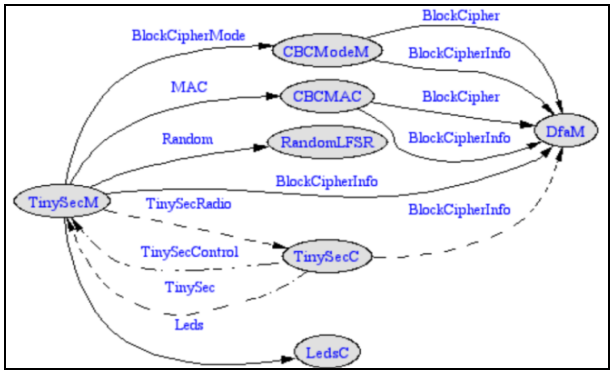


Fig 3: Flow Graph of Dfa

c. Code Size on different platforms

Table 3: DF Code Size on Different Platforms

DF		
Platforms	ROM	RAM
Pc	80384	1492880
Mica	18314	961
Mica2	18122	1006
Mica2dot	17952	1006

C. Castelluccia-Mykletun-Tsudik (CaMyTs) Algorithm

Castelluccia, Mykletun, and Tsudik propose a simple and provably secure additively homomorphic stream cipher that allows efficient aggregation of encrypted data. The main idea of the scheme is to replace the exclusive-OR (XOR) operation typically found in stream ciphers with modular addition (+). Since this new cipher only uses modular additions (with very small moduli), it is very well suited for CPU-constrained device.[8]

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

Castelluccia, Mykletun, Tsudik (CaMyTs) algorith[8][7]

Parameter: *select large integer* M

Encryption: *Message* $m \in [0, M-1]$,
randomly generated keystream $k \in [0, M-1]$
$c = (m + k) \bmod M$

Decryption: $Dec(c, k, M) = c - k (\text{mod } M)$

Aggregation: Let $c_1 = Enc(m_1, k_1, M)$ and
$c_2 = Enc(m_2, k_2, M)$
For $k = k_1 + k_2$, $Dec(c_1 + c_2, k, M) = m_1 + m_2$

It is assumed that 0 ≤ m <M. Due to the commutative property of addition, the above scheme is additively homomorphic. In fact, if $\mathbf{c_1}$ = **Enc** ($\mathbf{m_1}$, $\mathbf{k_1}$,**M**) and $\mathbf{c_2}$ = **Enc**($\mathbf{m_2}$,$\mathbf{k_2}$,**M**), then $\mathbf{c_1 + c_2}$ = **Enc**($\mathbf{m_1 + m_2}$,$\mathbf{k_1 + k_2}$,**M**).

Note that if n different ciphers **ci** are added, then M must be larger than $\sum_{i=1}^{n} m_i$; otherwise, correctness is not provided. In fact, if $\sum_{i=1}^{n} m_i$; mi is larger than M, decryption will result in a value m' that is smaller than M. In practice, if **p = max(mi)**, then M should be selected as $\mathbf{M = 2^{[\log_2(p*n)]}}$.

The key stream k can be generated by using a stream cipher, such as RC4, keyed with a node's secret key si and a unique message ID. This secret key is precomputed and shared between the node and the sink, while the message ID can either be included in the query from the sink or it can be derived from the time period in which the node is sending its values in (assuming some form of synchronization)[8].

a. Implementation

In this section, we present an implementation of the CAM algorithm by mathematical equations.

1) Key generation:

a) As shown in the algorithm select large integer M. Suppose we have chosen M=100000000

b) Now generate key stream k ϵ [0 , M – 1] randomly. Take K=5.

2) Encryption:

a) We have to enter the plain text *m* for encryption purpose. Suppose Plain Text is m = 251

b) Using the equation c = (*m* + k) mod M we can get the cipher text as below:
Cipher Text ***c*** = (*m* + *k*) mod M
= (251 + 5) mod 100000000 = **256**

3) Decryption:

a) the plain text can be recovered as
Plain Text ***m*** = (c - k) mod M
= (256 - 5) mod100000000 = **251**

b. Flow Graph

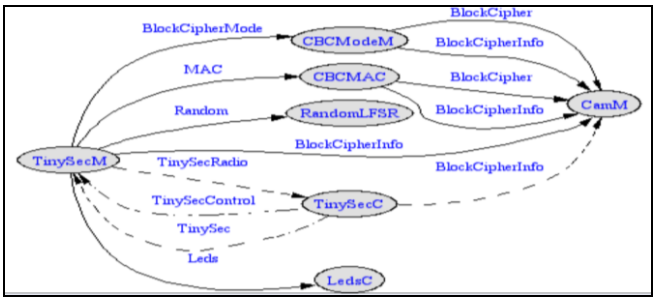


Fig 4: Flow Graph of Cam

c. Code Size on different platforms

CAM		
Platforms	**ROM**	**RAM**
Pc	77824	1288880
Mica	15094	807
Mica2	15118	852
Mica2dot	14948	852

Table 4: CAM Code Size on Different Platforms

D. CaMyTs+Domingo-Ferrer(CaMyTs/DF) Algorithm

The DF/CaMyTs combination is algebraically sound since CaMyTs as E1 encryption maps the plaintexts that are in $\mathbf{Z}_n$ E1 : $\mathbf{Z}_n$ → $\mathbf{Z}_n$, and DF uses the resulting cipher texts for its encryption E_2 : $\mathbf{Z}_n$ →**C**, while **C** is a usual known DF cipher text[8].

CaMyTs + Domingo-Ferrer (CaMyTs/DF) algorithm [8].

Parameter: *public key: large integer* M, $d \geq 2$
secret key: g *that divides* M; r *so that* r^{-1} exists in $\mathbb{Z}_M$

Encryption: *randomly generated keystream* $k \in [0, M-1]$
$e_1 = (k + m) \bmod M$
split e_1 into d parts $m_1 \ldots m_d$ that $\sum_{i=1}^{d}(m_i) \bmod g = e_1$ $C = [c_1, \ldots, c_d] = [m_1 r \bmod M, m_2 r^2 \bmod M, \ldots, m_d r^d \bmod M]$

Aggregation: scalar addition modulo M (like DF)

Decryption: $d_1 = (c_1 r^{-1} + \ldots + c_d r^{-d}) \bmod g$
$m = (d_1 - k) \bmod M$
where k is the sum of aggregated keystreams

a. Algorithm Implementation

In this section, we present an implementation of the DFCAM algorithm by mathematical equations.

1) Key generation:

a) Public key: Select large integer M. and d ≥ 0.
M = 20000000.

b) Secret key: Choose r and g such that **g** can divide **M** and $\mathbf{r^{-1}}$ must be in $\mathbf{Z_M}$.
r = 2
g = 10000000

c) randomly generate key stream k ϵ [0,M – 1].K = 3

2) Encryption:

a)We have taken Plain text ***m*** **= 5**

b) Calculate e1 = (k + *m*) mod M = (3 + 20000000) mod 10000000 = 8

c)split e1 into d parts m1 . . .md that
$\sum_{i=1}^{d}(m_i) \bmod$ g = e1 * C = c[c_1, . . . , cd] =
[m1*r mod M,m2 * r^2 mod M,.., md * r^d mod M]
here, value of e is 8 which is Even no.So,we have to split it in to Two part of plain text: 2,6
m1=2
m2=6

d) Cipher Text **c** = c1 = m1 * r mod M = 4
c2 = m2 * r^2 mod M = 24

Published By:
Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP)

3) Decryption:

a)For finding the plain text first of all we have to find d1 using the equation,

$$d_1 = (c_1 r^{-1} + \ldots + c_d r^{-d}) \bmod g$$

$d11 = c_1 * r^{-1}=2$

$d12 = c_2 * r^{-2} = 6$

So, value of d1 = d11 + d12 = 8

b) We can get the Plain text bake from cipher text as shown below:

Plain text ***m*** = (d – k) mod M = **5** , where k is the sum of aggregated key stream.

b. Flow Graph

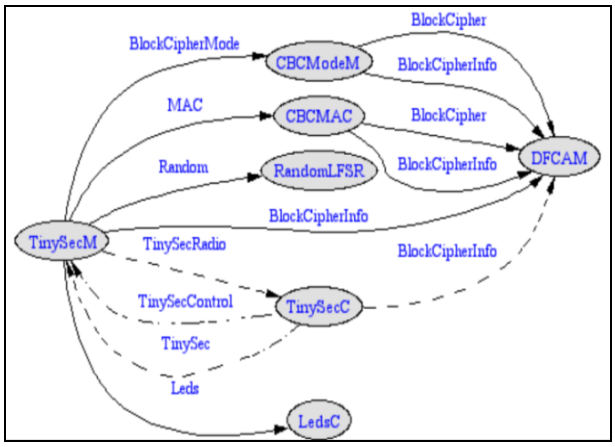


Fig 5: Flow Graph of DFCAM

c. Code Size on different platforms

DF+CAM		
Platforms	**ROM**	**RAM**
Pc	80384	1500880
Mica	18534	969
Mica2	18304	1014
Mica2dot	18134	1014

Table 5: DF+CAM Code Size on Different Platforms

IV. IMPLIMENTATION METHODOLOGY

A. Tools used

The following major tools were used for implementing the algorithm and simulating the sensor network.

B. NesC (Network Embedded System C)

NesC is a C dialect. Program structure is the most essential and obvious difference between C and nesC. C programs are composed of variables, types, and functions defined in files that are compiled separately and then linked together. NesC programs are built out of components that are connected (“wired”) by explicit program statements; the nesC compiler connects and compiles these components as a single unit. The nesC compiler can take advantage of this explicit wiring to build highly optimized binaries. Current implementations of the nesC compiler (nesc1) take nesC files describing components as input and output a C file. The C file is passed to a native C compiler that can compile to the desired microcontroller or processor. Figure shows this process. The nesC compiler carefully constructs the generated C file to maximize the optimization abilities of the C compiler. For example, since it is given a single file, the C compiler can freely optimize across call boundaries, in lining code whenever needed. The nesC compiler also prunes dead code which is never called and variables which are never accessed: since there is no dynamic linking in nesC, it has a complete picture of the application call graph. This speeds the C compilation and reduces program size in terms of both RAM and code.

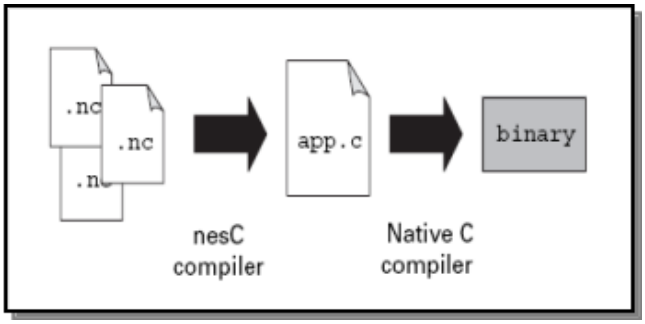


Fig 6: The nesC compilation model. The nesC compiler loads and reads in nesC components, which it compiles to a C file. This C file is passed to a native C compiler, which generates a mote binary.

Table 6 summarizes the difference in how programs are structured in C, C++ and nesC. In C, the typical high-level programming unit is the file, with an associated header file that specified and documents the file’s behavior. The linker builds applications out of files by matching global names; where this is not sufficient to express program structure (e.g. for callbacks), the programmer can use function pointers to delay the decision of which function is called at what point.

Structural Elements	C	C++	NesC
Program unit	File	Class	Component
Unit specification	Header file	Class declaration	Component specification
Specification Pattern	-	Abstract class	Interface
Unit composition	Name matching	Name matching	Wiring
Delayed composition	Function pointer	Virtual method	Wiring

Table 6: Program Structure in C, C++ and nesC

C++ provides explicit language mechanisms for structuring programs: classes are typically used to group related functionality, and programs are built out of interacting objects (class instances). An abstract class can be used to define common class specification patterns (like sending a message); classes that wish to follow this pattern then inherit from the abstract class and implement its methods – Java’s interfaces provide similar functionality. Like in C, the linker builds applications by matching class and function names. Finally, virtual methods provide a more convenient and more structured way than function pointers for delaying beyond link-time decisions about what code to execute.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

In nesC, programs are built out of a set of cooperating components. Each component uses interfaces to specify the services it provides and uses; the programmer uses wiring to build an application out of components by writing wiring statements, each of which connects an interface used by one component to an interface provided by another. Making these wiring statements explicit instead of relying on implicit name matching eliminates the requirement to use dynamic mechanisms (function pointers, virtual methods) to express concepts such as callbacks from a service to a client.

A nesC program is a collection of components. Every component has a signature, which describes the functions it needs to call as well as the functions that others can call on it. A component declares its signature with interfaces, which are sets of functions for a complete service or abstraction. There are two kinds of components: modules and configurations. Modules are components that implement and call functions in C-like code. Configurations connect components into larger abstractions. Modules and configurations can be used interchangeably when combining components into larger services or abstractions. The two types of components differ in their implementation sections. Module implementation sections consist of nesC code that looks like C. Module code declares variables and functions, calls functions, and compiles to assembly code. Configuration implementation sections consist of nesC *wiring* code, which connects components together. Interfaces describe a functional relationship between two or more different components. The role a component plays in this relationship depends on whether it provides or uses the interface. Like components, interfaces have a one-to-one mapping between names and files: the file. An interface declaration has one or more functions in it. Interfaces have two kinds of functions: ***commands*** and ***events.*** Whether a function is a command or event determines which side of an interface – a user or a provider – implements the function and which side can call it. Users can ***call*** commands and providers can ***signal*** events. Conversely, users must implement events and providers must implement commands.

C. Tiny OS

TinyOS is a lightweight operating system specifically designed for low-power wireless sensors. TinyOS differs from most other operating systems in that its design focuses on ultra low-power operation. Rather than a full-fledged processor, TinyOS is designed for the small, low-power microcontroller's motes have. Furthermore, TinyOS has very aggressive systems and mechanisms for saving power. TinyOS makes building sensor network applications easier. It provides a set of important services and abstractions, such as sensing, communication, storage, and timers. It defines a concurrent execution model, so developers can build applications out of reusable services and components without having to worry about unforeseen interactions. TinyOS runs on over a dozen generic platforms, most of which easily support adding new sensors. Furthermore, TinyOS's structure makes it reasonably easy to port to new platforms. TinyOS applications and systems, as well as the OS itself, are written in the nesC language. nesC is a C dialect with features to reduce RAM and code size, enable significant optimizations, and help prevent low-level bugs like race conditions.

D. What TinyOS Provide

At a high level, TinyOS provides three things to make writing systems and applications easier:

a. A component model, which defines how you write small, reusable pieces of code and compose them into larger abstractions;
b. A concurrent execution model, which defines how components interleave their computations as well as how interrupt and non-interrupt code interact;
c. Application programming interfaces (APIs), services, component libraries and an overall component structure that simplify writing new applications and services.

The component model is grounded in nesC. It allows you to write pieces of reusable code which explicitly declare their dependencies. For example, a generic user button component that tells you when a button is pressed sits on top of an interrupt handler. The component model allows the button implementation to be independent of which interrupt that is – e.g. so it can be used on many different hardware platforms – without requiring complex callbacks or magic function naming conventions.

The concurrent execution model enables TinyOS to support many components needing to act at the same time while requiring little RAM. First, every I/O call in TinyOS is *split-phase*: rather than block until completion, a request returns immediately and the caller gets a callback when the I/O completes. Since the stack isn't tied up waiting for I/O calls to complete, TinyOS only needs one stack, and doesn't have threads. Any component can post a task,(which are lightweight deferred procedure calls) which TinyOS will run at some later time. Because low-power devices must spend most of their time asleep, they have low CPU utilization and so in practice tasks tend to run very soon after they are posted (within a few milliseconds). Furthermore, because tasks can't preempt each other, task code doesn't need to worry about data races. Low-level interrupt code (discussed in the advanced concurrency can have race conditions, of course: nesC detects possible data races at compile-time and warns you. Finally, TinyOS itself has a set of APIs for common functionality, such as sending packets, reading sensors, and responding to events. In addition to programming interfaces, TinyOS also provides a component structure and component libraries. For example, TinyOS's Hardware Abstraction Architecture (HAA), defines how to build up from low-level hardware (e.g. a radio chip) to a hardware-independent abstraction (e.g. sending packets). Part of this component structure includes resource locks which enable automatic low-power operation, as well as the component libraries that simplify writing such locks. TinyOS itself is continually evolving. Within the TinyOS community, "Working Groups" form to tackle engineering and design issues within the OS, improving existing services and adding new ones. TinyOS has a set of standard, stable APIs for core abstractions, but this set is always expanding as new hardware and applications emerge. The best way to stay up to date with TinyOS is to check its web page www.tinyos.net and participate in its mailing lists. The website also covers advanced TinyOS and nesC features which are well beyond the scope of this book, including binary components, over-the-air reprogramming services, debugging tools, and a nesC reference manual.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

V. CONCLUSION

Targeted at TinyOS, TinyECC is written in nesC, with occasional in-line assembly code to achieve further speedup for popular sensor platforms includingMICAz, TelosB, Tmote Sky, and Imote2. Different combinations of optimizations have different ROM/RAMconsumption, execution time, and energy consumption. This gives the developers great flexibility in integrating TinyECC in their applications.

With different combinations of activated optimizations. To understand the impact of each optimization technique, we compare the execution time, ROM/RAM consumption, and energy consumption with and without the given optimization enabled on MICAz [2], Tmote Sky [6], and Imote2 [1]. In addition, our experiments also present the performance results and the resource usages for the most computationally efficient configuration (i.e., fastest execution and least energy consumption) and the most storage-efficient configuration (i.e., least ROM and RAM usage) of TinyECC on these common sensor platforms, respectively.

Plateform PC:

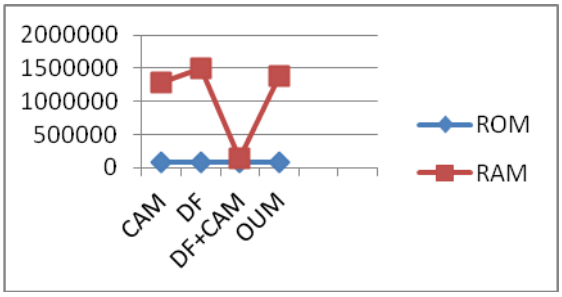


Fig 7: Chart for PC Plateform

Plateform mica:

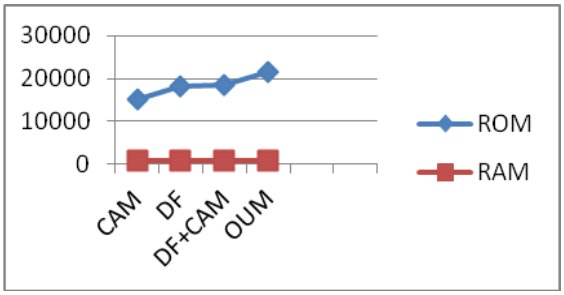


Fig 8:chart for mica Plateform

Plateform mica2:

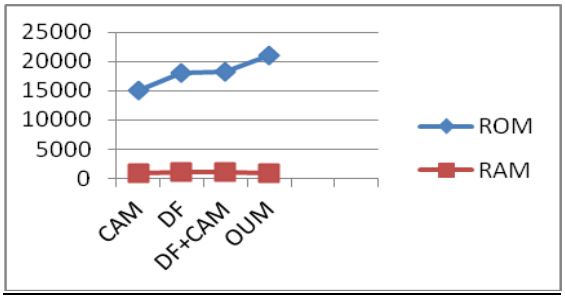


Fig 9: Chart for Mica2 Plateform

Plateform mica2dot:

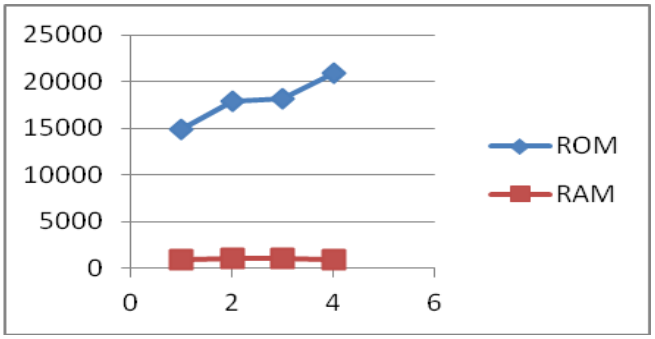


Fig 10: Chart for Mica2dot Plateform

From above implementation of all the algorithms we will analyze that each a provides the best ratio between encryption and decryption costs. This comes at the cost of a bigger cipher text size, which we still consider acceptable for applications which require only seldom polling and aggregation of the values.

ACKNOWLEDGMENT

A Special Thank To Prof. Vivaksha Jariwala For Giving His Guidance And Support To Carry Out This Reserch And Make This Reserch Paper.

REFERENCES

1. Jason Lester Hill: "System Architecture for Wireless Sensor Networks", Doctor of Philosophy In Computer Science In The Graduate Division Of The University Of California, Berkeley
2. Steffen Peter, Dirk Westhoff, Member, IEEE, and Claude Castelluccia: "A Survey on the Encryption of Convergecas Traffic with In-Network Processing".
3. Suat Ozdemir , Yang Xiao : "Computer Networks", Department of Computer Science, The University of Alabama, Tuscaloosa, AL 35487-0290, United States.
4. Tatsuaki Okamoto, Shigenori Uchiyama: "A New Public Cryptosystem as secure as factoring", Florida Atlantic University, Boca Raton, FL, USA
5. Joao Girao and Dirk Westhoff NEC Europe Ltd. Kurf¨ursten-Anlage "Public Key Based Cryptoschemes for Data Concealment in Wireless Sensor Networks", Einar Mykletun Computer Science Department University of California, Irvine.
6. D. Naccache and J. Stern. A New Public Key Cryptosystem Based on Higher Residues. ACM Conference on Computer and Communications Security, pages 59–66, 1998.
7. Einar Mykletun:"Public Key Based Cryptoschemes for Data Concealment in Wireless Sensor Networks", Computer Science Department University of California, Irvine
8. Steffen Peter, Dirk Westhoff, Member, IEEE, and Claude Castelluccia:" A Survey on the Encryption of Convergecast Traffic with In-Network Processing Wireless Sensor Network Security: A Survey.".
9. An Liu:" A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks", Department of Computer Science NC State University, Raleigh, NC 27695 email: aliu3@ncsu.edu
10. Chris Karlof, Naveen Sastry, David Wagner:" A Link Layer Security Architecture for Wireless Sensor Networks" ckarlof@cs.berkeley.edu ,UC Berkeley .
11. C. Castelluccia, E. Mykletun, and G. Tsudik, "Efficient Aggregation of Encrypted Data in Wireless Sensor Networks," Proc. Second Ann. Int'l Conf. Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous '05), July 2005.
12. J. Domingo-Ferrer, "A Provably Secure Additive and Multiplicative Privacy Homomorphism," Proc. Fifth Information Security Conf. (ISC '02), pp. 471-483, 2002.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)