# ECE 493 Final Report
# Energy and Power Comparison of Cryptographic Algorithms on Sensor Nodes

Kiran Rizvi, John Pham, Mark McDermott, and Reginald Brown
Advisor : Dr. Jens-Peter Kaps

***Abstract*—We examine the feasibility of adding dedicated encryption hardware to low-power wireless sensor nodes. Having the ability to encrypt communications between sensor nodes in a network is a desirable feature for obvious reasons. The question is: What is the best way to approach adding encryption capabilities? We examine two main approaches and consider the tradeoffs for each approach. Adding hardware will increase the speed at which encryption can be performed and will free up processing resources on the sensor node. But extra hardware will increase the overall power consumption of system. Another disadvantage of dedicated hardware is that I/O pins on the sensor node that could be used for sensor applications need to be reserved for communication between the node's processor and the encryption hardware. On the other hand, if all the encryption is performed in software on the sensor node then any encryption schemes that benefit from parallizable hardware will certainly run slower than is necessary. As a result, power consumption is higher and processing time is longer than required. In this project we attempt to quantify these tradeoffs so that an informed decision can be made when designing a secure sensor node system. We focus primarily on power consumption but will discuss the other tradeoffs that have been mentioned.**

## CONTENTS

## I. INTRODUCTION

### A. Statement of Need

WIRELESS mesh sensor networks are commonly used to gather commercial and military intelligence as well as for industrial monitoring. It is important that the individual nodes have a security system to prevent unauthorized users from controlling them. Also, in some applications the data being collected must be transmitted to an authorized controlling station securely. Knowing the effect that adding encryption capabilities to a sensor node will have on power consumption and general usability of the node is invaluable information. Thus, in this project, we gather power consumption data for a variety of encryption implementations and compare them to each other to provide said information for future development.

## B. Existing Work

Our work is extending the previous work of the ECE492-493 who conducted the project [1]. Their work included the design of system that allowed a MICA2 sensor node and an FPGA based implementation of AES(ref) to communicate via UART. With the addition of a second sensor node attached to a PC they were able to control the system and retrieve encrypted bytes from the sensor node which had the attached hardware encryption solution. Their results showed the viability of such a system.

The previous team also designed a FPGA platform that connects directly to the sensor node through a hirose connector. At the time that it was created there was no consideration of alternate communication methods between the FPGA and sensor node. As a result the custom PCB that they designed only supports the UART connection.

## C. Design Trade-offs

Each of the three encryption schemes are tested in two different ways to show the tradeoffs involved with adding encryption to a sensor node. First, we use the sensor node and the FPGA communicating through the parallel interface. This is the fastest mode of communication which means that the FPGA can remain idle (using less energy) more. However the parallel interface uses nine of the very precious I/O pins on the sensor node.

The final solution is to not use the FPGA at all and perform all the encryption directly on the sensor node. This means that no power needs to be supplied to external hardware and no I/O pins are used for encryption. However, many clock cycles are used to perform the encryption which results in a higher percentage of the sensor node's total energy usage being devoted to encryption.

## D. Proposed Solution

We extend the previous team's work by increasing the number of implemented encryption standards to three: AES, XTEA, and PRESENT. With the exception of PRESENT all of the hardware implementations of these standard were already implemented at the time this project began. We implement software version of each of these standards as well. We also add a parallel interface standard for communication between the sensor node and FPGA.

The power consumption of each encryption standard will be measured for the cases of FPGA for encryption, and all encryption done on the sensor node. The results will be compared to determine which solution is the most energy efficient.

## E. Evaluation Criteria

When measuring the power consumption of the FPGA based implementations we split the data into sections corresponding to the following modes of operation: receiving data from the sensor node, encrypting, sending data back, and sitting idle. It is important to know the contribution of each mode because a real-world solution would include the ability to turn the encryption hardware on and off when it is not in use. We also, wish to know how much power is consumed during just the encryption cycle so that we can calculate what effect encrypting multiple blocks of data at a time has on overall energy efficiency.

All of the results obtained for the FPGA solutions will be adjusted according to power usage simulations for ASIC implementations of the same design. This is because the expected application of the research is towards an ASIC based add-on for sensor nodes. We are using the FPGA purely as a proof of concept.

The power measurements for the software-only encryption implementations will be taken in a similar manner FPGA measurements. Since operating modes of the sensor node do fall neatly into the above categories (receive,encrypt,transmit) we can not perform a direct comparison between them. We can however compare the total energy used during a complete cycle of data acquisition, data encryption, and transmission to base station PC.

The solution which achieves a balance between usability (see design trade-offs) and energy efficiency will be marked as the most viable means for adding encryption to wireless sensor nodes.

## F. Team Responsibilities

| | |
|---|---|
| Kiran Rizvi | Project Manager, Hardware Encryption Implementation, Hardware Integration |
| John Pham | Parallel interface (Hardware/Software), Software Integration, Node to Basestation Communication |
| Mark McDermott | Hardware Integration, Software Encryption Implementation, Power Measurements |
| Reginald Brown | Hardware Integration |

## II. Technical Section

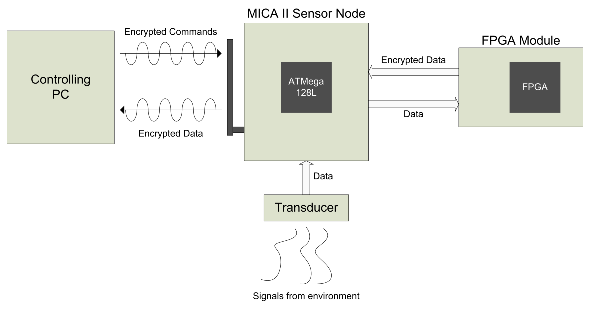

Fig. 1. High level view of system components.

The overall system shown in Figure 1 works as follows. The sensor node gathers data from the environment. This data is either stored or transmitted based on commands from the controlling base station. When commanded to, the sensor node passes the collected data to the encryption module where it is encrypted and then sent back to the sensor node. In a production system the sensor node would control when the encryption module is powered. For our system we simply power the module all the time. Once the data is encrypted it will be sent over the radio to the base station computer where it can be decrypted and processed.

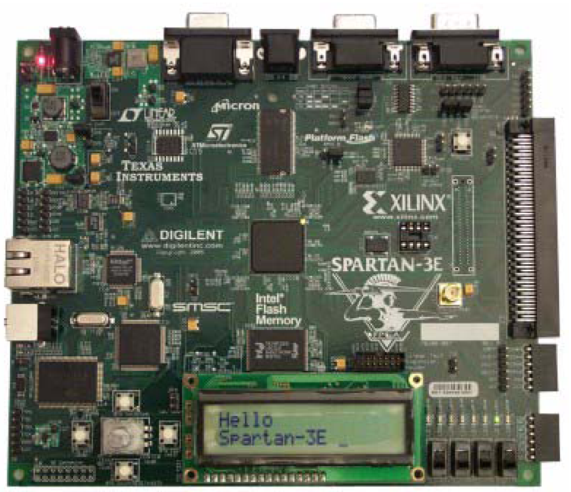

Fig. 2. Spartan3E development board from Digilent.

The software only implementation has an identical interface to the base station but does all data encryption directly on the sensor node in software.

Fig. 3. Crossbow Mica2 sensor node.

NOTE: This system is only a testbed to allow us to compare the energy efficiency of the encryption algorithms. We could have done these tests on any small, low-power platform but because of their availability and ease of use we choose the Mica2 sensor node (Figure 3) and the Digilent Spartan3E development board (Figure 2) as our testing platform.

## A. Parallel Interface

The parallel interface is designed to use as few pins as possible. That said, it is still using 9 out of the 50 available pins on the sensor node's microcontroller. (There are fewer pins actually available due to pins already being assigned to the radio, power supply control, status LEDs, etc.) The interface uses 8 data lines and a single control/clock line.

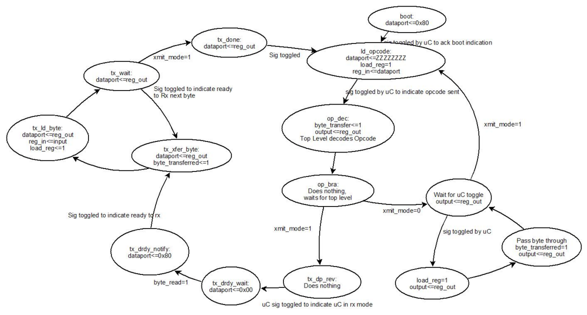

Fig. 4. State Transition Diagram of Communication Module for FPGA Side

After reset, the interface is in send mode (sensor node transmits to FPGA). When ready to send data, the first byte is written to the databus and then the signal is toggled. The first byte sent is always a command word which tells the main FPGA control (external to the parallel communication module, see Figure 5) what to expect from the sensor node. In response to this `xmit_mode` is set, which the communication module will respond to. If `xmit_mode` is low, the FPGA waits for data (and a signal line toggle) repeatedly until `xmit_mode` goes high.

When this happens, the communication module goes to idle and waits for another command word. This may be ignored by the FPGA top level control holding `xmit_mode` high to enter the transmit states. In the transmit modes, the communication module will begin handshaking with the sensor node to change the direction of the port. The communication module will wait until the signal line is toggled a second time which indicates that the sensor node has set its port to high impedance. Then the communication module writes `0x80` to the port. This indicates to the sensor node that the FPGA is ready to send data. The sensor node will toggle the signal line every time that it is ready for another byte. This works because the FPGA is clocked at 50MHz and the microcontroller on the sensor node is clocked at 8MHz. The FPGA is always ready to send another byte by the time that the sensor node toggles the signal line again. When `xmit_mode` goes high again and the signal line is toggled again to acknowledge receipt of the last byte, the FPGA goes back into idle state waiting for a command word.

## B. TinyOS

TinyOS [2] is used in order to abstract some of the hardware-specific features of the MICA2, provide drivers to the hardware, and to allow for low-overhead multitasking. In order to program for TinyOS, a language called NesC must be used, which adds event-based programming and split-phase features for non-blocking calls, as well as interfaces to C. TinyOS uses a form of cooperative multitasking, requiring large functions to be split up into tasks that are scheduled.

In NesC, there are multiple types of functions: events, tasks, and commands. Commands and events are defined inside interfaces. They behave like regular functions, however a command is called to signify to a module that it should do something and is defined within the module where it is declared, while an event is defined outside and is called to signify that a response should occur. Both are blocking calls. Tasks are void functions with no parameters that are to be scheduled to run at some point in the future.

## C. Communication With Basestation

Communication with the basestation PC is done through another sensor node which is attached via a USB serial port to the computer. The features of the application are geared towards verifying the correct encryption and transmission of data. To operate the system, the operator sends the key and the plaintext to the node. The node encrypts and immediately returns the ciphertext. Due to the lack of flow control and reliability in the TinyOS radio stack, stop-and-wait was implemented on the application level.

*1) BaseStation Application:* The basestation application is written in java, and reads in a file containing hex strings, transmitting the first line as key and subsequent lines as blocks of data to be encrypted, splitting the data into 64 bit segments. It transmits these blocks in reversed order with a sequence number counting down. It waits for an ACK after each packet containing the sequence number of the last transmitted packet before continuing to transmit. After a timeout period without a valid ACK it will attempt to retransmit, retrying 10 times. After transmission is done, it then waits for encrypted packets to be received. When this happen, an ACK packet with the sequence number of the last received packet is sent and the data inside the packets are displayed.

*2) Radio:* The radio code sends out an ACK with the sequence number of the last packet received for both key and data packets. TinyOS allows different packet types, and each packet type has it's own handler. The key handler places the contents of the packet into a key buffer, and the data handler places it in a location in the buffer determined by the sequence number multiplied by the packet data size. Bounds checking has not been implemented, therefore this is susceptible to buffer overflow attacks. When the key and the data packet with the sequence number 0 is received, the mote begins encrypting the data. Data is sent and received in reverse order to avoid having to transmit a done command or the length of the data, since the first and last sequence number alone is sufficient to determine this. This is not suitable for reliability protocols other than stop and wait due to packet ordering. When the data is encrypted, the encrypted data is transmitted in a fashion similar to that of the the base station transmitting.

## D. Software

*1) Software Encryption Modules:* All of the encryption modules implemented in software were based on source code freely available on the internet. [3] [4] [5] The source code obtained from these sources was modified for speed, oriented to 8 bits, and conformed to our TinyOS interfaces.

*2) Encryption Interface:* The software interface for the crypto modules is defined in Listing 1. This interface is implemented by both the software and hardware encryption modules. The hardware driver modules also implement the `SplitControl` [6] interface to turn the FPGA on and

```
/**
 * Interface implemented by encryption modules
 */
interface Crypt{
    /**
     * Loads crypto module w/ key. In some implementations, key size is already
     * known so size is ignored. Size is specified in bytes.
     *
     * Precondition: Module must be initialized, and no other operation on
     * crypto module already pending
     *
     * Postcondition: Key loaded when keyLoaded() fires
     *
     * @param key Key to use when encrypting
     * @return 1 when success, 0 when failed.
     */
    command error_t setKey(uint8_t* key);

    /**
     * Encrypts data w/ key loaded in module. Length of datain and dataout must
     * be identical and be an integer multiple of the block size of the crypto
     * implementation.
     *
     * Precondition: key must already be loaded, and no encryption already
     * pending. keyLoaded() must be fired.
     *
     * Postcondition: dataOut is filled with
     * encrypted data
     *
     * @param dataIn Pointer to data to encrypt
     * @param dataOut Pointer to buffer to write encrypted data to.
     * @param numBlocks Size of data in blocks.
     * @return 1 when success, 0 when failed
     */
    command error_t encryptData(uint8_t* dataIn, uint8_t* dataOut,
            uint16_t numBlocks);

    /**
     * Returns block size in bytes
     * @return Block size
     */
    command uint8_t getBlockSize();

    /**
     * Returns key size in bytes
     * @return Key size
     */
    command uint8_t getKeySize();

    /**
     * Fired when key is loaded into crypto module
     * @param error SUCCESS if loading key successful, FAIL otherwise
     */
    event void keyLoaded(error_t error);

    /**
     * Fired when data encrypted
     * @param error SUCCESS if encryption successful, FAIL otherwise
     */
    event void dataEncrypted(error_t error);
}
```

Listing 1. Software interface for crypto modules in Crypt.nc

off. This is not actually implemented in any of the drivers (other than waiting for the FPGA to turn on) as we did not implement power control for the FPGA. The software implementations also implement the `SplitControl` interface with dummy functions in order to minimize required changes. In order to switch between modules, the `ENC_IMPL` macro is set to algorithms defined in `HWCryptC.nc` and in `MoteEncryption.h`.

*3) Communications Interface:* The interface in Listing 2 is implemented by `MoteRadioC` in order to transmit data coming in and out of the sensor node. The operation of this code is described in Section II-C2. Prior to the use of `MoteRadioC` the toplevel mote code must initialize the radio hardware by utilizing the `SplitControl` interface provided by `ActiveMessageC` [6].

## E. FPGA Encryption Cores

The FPGA design was designed to be as generic and reusable as possible. To do this we defined a standard interface between the four main components: encryption core, key store (RAM), communication module, and top level control.(see Figure 5) The key store is used to keep the key intact so

```
/**
 * Interface for transmitting data in and out.
 * of mote
 */
interface KeyDataXmit{
    /**
     * Precondition: Implementation specific
     * Postcondition: sendDataDone() called at some point in the future
     * @param data Pointer to data to be sent
     * @param len Size of data in number of datapacket data size chunks,
     * implementation specific.
     */
    command error_t sendData(uint8_t *data,uint8_t len);

    /**
     * Precondition: Transmission of data complete
     * Postcondition: User defined
     * @param err Status of data transmission, may return SUCCESS, FAIL, or
     * EBUSY
     * @return SUCCESS or FAIL
     */
    event error_t sendDataDone(error_t err);

    /**
     * Precondition: None
     * Postcondition: Needs unlock() called to receive data
     * Sets address of receive key and data buffers
     * TODO Add bounds limits
     * @param keyAddr Address of key RX buffer
     * @param dataAddr Address of data RX buffer
     */
    command error_t setAddresses(uint8_t *keyAddr,uint8_t *dataAddr);

    /**
     * Precondition: Key successfully received after setAddresses() and unlock()
     * Postcondition: None
     * @return SUCCESS if succeeded
     * @param len: Length of key in keypacket data size chunks
     * @return SUCCESS or FAIL
     */
    event error_t keyLoaded(error_t err,uint8_t len);

    /**
     * Precondition: Data successfully received after setAddresses() and unlock()
     * Postcondition: None
     * @param err: SUCCESS if succeeded
     * @param len: Length of data in datapacket data size chunks
     * @return SUCCESS or FAIL
     */
    event error_t dataLoaded(error_t err,uint8_t len);

    /**
     * Precondition: Object is previously unlocked
     * Postcondition: Radio cannot receive key or data
     * @return SUCCESS or FAIL
     */
    command error_t lock();

    /**
     * Precondition: Object is previously locked
     * Postcondition: Radio can receive key or data
     * @return SUCCESS or FAIL
     */
    command error_t unlock();
}
```

Listing 2. Software interface for crypto modules in Crypt.nc

that when encrypting more that one block of data we need only transmit the key once.

Both methods of communicating with the sensor node (parallel and serial) have the same top level structure. They differ only in the number and type of connections (See section II-A ) to the sensor node. The communication modules serve as translators between the internal top level signals and the signals coming from and going to the sensor node. The internal signals are defined in Table I

The basic operation is as follows. When a byte has been fully transferred from the sensor node then the ByteTransferred signal is asserted. This signals the control module to allow the received data onto the bi-directional bus into the encryption core. In general the first byte that is transferred is not read by the encryption core because it is a command byte that tells the control module how to interpret the remaining data. After reading the command byte, Control manages each incoming byte from the communication module choosing to whether or not to save it in RAM (key or text) and when to enable the encryption. When all the data and key are received then encryption is performed. When the encryption core has finished with a byte, it informs Control which then manages the passing of data out of the encryption core and through the communication module back to the sensor node.

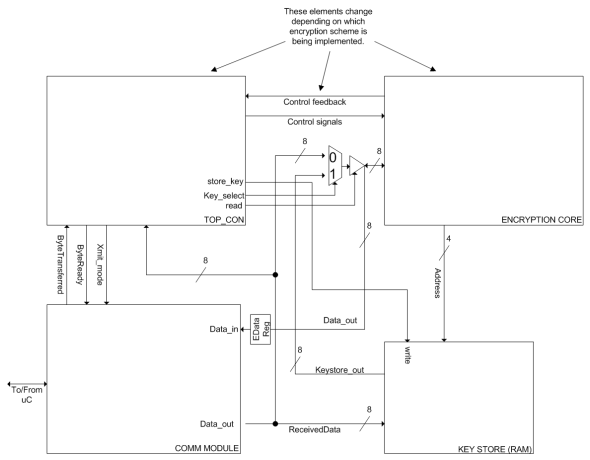

Fig. 5. Block Diagram of top level FPGA design

All of the encryption schemes that we implemented operate this way with slight variations. We attempted to keep them as similar as possible so that power consumption can be meaningfully compared.

*1) AES:* The AES implementation takes 128 bits of data and 128 bits of key. These bytes come in interleaved starting with a byte of key first. The top level control can operate it two different modes. The first is where all of the key and data are read. The key will automatically be stored in the key store RAM. In the second mode, only the data is sent from the sensor node and the key bytes will be read from the RAM. The mode is indicated by the two most significant bits of the first byte sent (command byte): ”00” for key and data, ”01” for just data.

Every time the communication module receives a byte from the sensor node, the control will decide if that byte is key or data. For AES 32 bytes are sent in total which come in interleaved fashion starting with a byte of key. Every other byte is stored in the keystore RAM. If a byte is data, then AES is enabled for one clock cycle to allow it to read in the byte. When all bytes have been received AES is enabled for all clock cycles and encryption is performed (the communication module switches directions during this time). When encryption of a byte is done, `ByteEncrypted` is set high which signals control to stop AES and wait for the communication module to send the byte to the sensor node. Once it is sent, then encryption resumes until another byte is ready to be sent back. This process continues until all the encrypted bytes have been sent back to the sensor node. At this point control returns to idle and `reset_AES` is asserted. See Table II for a summary of AES specific control signals.

*2) Present:* Present uses the key store differently from AES because it is capable of reading all 128 bits of key data at once. Therefore the keystore is a large register instead of a RAM. To start an encryption cycle for Present the sensor node first sends 16 bytes of key and 8 bytes of data. The top level control in the FPGA will save the key into the key store and pass the

TABLE I
FPGA INTERNAL COMMUNICATION SIGNALS

| Type | Name | Active | Width | Duration | Purpose |
|---|---|---|---|---|---|
| From Control | `ByteReady` | high | 1 | 1 clk | Signals that `Data_in` is valid |
| | `xmit_mode` | high | 1 | varies | Tells communication with uC to change directions |
| To Control | `ByteTransfered` | high | 1 | 1 clk | Tells control that a byte has been transmitted (either in or out) |
| From Datapath | `Data_in` | n/a | 8 | held | Encrypted data to be sent out |
| To Datapath | `Data_out` | n/a | 8 | held | Data received from uC |

TABLE II
AES CONTROL SIGNALS

| Type | Name | Active | Width | Duration | Purpose |
|---|---|---|---|---|---|
| control | `EnableECore` | high | 1 | held | Allows AES to run |
| | `oe` | high | 1 | held | Allows output on bidirectional if `clk` is high |
| | `reset_AES` | high | 1 | held | Resets AES |
| | `transmit` | high | 1 | held | Gives AES complete control of bidirectional bus |
| Feedback | `ByteEncrypted` | high | 1 | 1 clk | Signals that a byte has been encrypted |

data to Present. Then after all the data has been sent, Present is enabled at which time the entire key is read from RAM in a single operation.

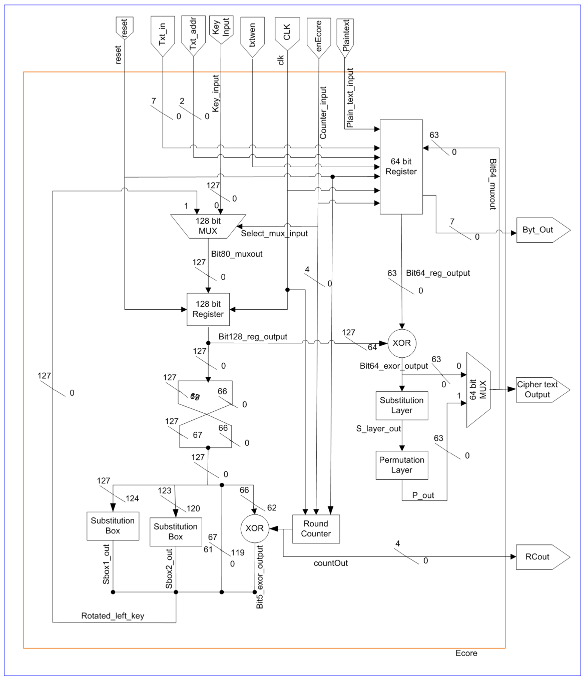

Fig. 6. Block diagram of PRESENT.

Once the encryption is finished, the encrypted data is sent back to the sensor node in the same manner as with AES.

*3) XTEA:* The operation of XTEA very similar to that of AES with two key differences. AES will perform the same operation each time it is sent data from the sensor node. commands are: load key, load data, and encrypt. The communication between the FPGA and sensor node is identical because the controller sends these commands to XTEA and enables operations at the appropriate times. In addition, the key is not used as scratch space, therefore having a seperate keystore external from the encryption core for quick reloading of the key is unnecessary.

The XTEA encryption core has also undergone changes during this project even though it was finalized prior to when we started the project. The capability to halt the encryption process has been added in order for core to wait upon more data coming from the sensor node.

### *F. Mote Based Encryption*

The software framework that controls communication with the FPGA is leveraged to make software implementations of all the encryption standards very easy to perform. For each of the encryption standards a TinyOS module is created that performs the encryption and conforms to the same interface definition has does the sensor node's side of the parallel communication standard.

The main controlling software on the sensor node, which handles radio communication and data acquisition is unaware of how encrypted data is being acquired. It simply passes data from the radio to the interface and reads encrypted data back regardless of what kind of encryption scheme is implemented (hardware or software).

## III. EXPERIMENTATION

### *A. Verification of Encryption Implementations*

*1) Hardware:* Each hardware implementation was verified by observing the data as it crossed the bus between the sensor node and the FPGA with a logic analyzer. Later when the radio component was finished we also verified the correct results by printing the received packets on the base station PC.

TABLE III
PRESENT CONTROL SIGNALS

| Type | Name | Active | Width | Duration | Purpose |
|---|---|---|---|---|---|
| Feedback | `RecievedData` | high | 2 | held | Opcode value coming from comm |
| | `roundCounter` | high | 5 | held | Counter for rounds of PRESENT |
| Control | `enEcore` | high | 1 | held | Starts encryption when high |
| | `writeKey` | high | 1 | held | Enables write to keystore |
| | `KeyAddr` | high | 4 | held | Address of key byte to write to |
| | `writeTxt` | high | 1 | held | Enables writing of text to text store |
| | `TxtAddr` | high | 4 | held | Address of text byte to write to |
| | `rstEcore` | high | 1 | held | Soft resets the encryption core |

TABLE IV
XTEA CONTROL SIGNALS

| Type | Name | Active | Width | Duration | Purpose |
|---|---|---|---|---|---|
| Feedback | `RecievedData` | high | 2 | held | Opcode value coming from comm |
| | `w` | high | 1 | held | Feedback from TEA control unit signifying TEA is writing on data bus |
| | `read` | high | 1 | held | Feedback from TEA control unit signifying TEA is reading data bus |
| Control | `stopEcore` | low | 1 | held | HALT COMMAND |
| | `stopKeyStore` | high | 1 | held | Enables/disables loading of key into TEA's internal keystore |
| | `command` | high | 2 | held | Command to TEA control unit- '00' read data command, '01' read key command, '10' encrypt command, '11' decrypt command |
| | `byteReady` | high | 1 | held | Signifies that bytes are ready to be transmitted to the microcontroller |
| | `encrypting` | high | 1 | held | Output register is loaded |
| | `waiting` | high | 1 | held | Output register loaded when `ByteTransferred` |
| | `idle_s` | high | 1 | held | If idle or reset=0, a soft reset is performed on TEA |
| | `oe` | high | 1 | held | Enables TEA output |

*2) Software:* The verification procedure for software was very simple; we just printed out the results of each radio transmission to the base station. In both cases we used test vectors and results obtained from reference software implementations of each algorithm and compared each byte with our results to verify their correct operation.

## B. Timing

The length of time needed for encryption was obtained by measuring the time difference between the start and stop of an encryption cycle on the logic analyzer. Also, we used this method to obtain time measurements for how long it took to perform each part of the cycle (send data to FPGA, encrypt, receive encrypted data). Of course, for the software only implementations there is no outside communication and therefore we only record the total time taken to encrypt a block (or two). All comparisons are made between encryption cycles that successfully encrypt 16 bytes in total. This means that, in the cases of XTEA and PRESENT, we encrypt two blocks and record the total time for both to be encrypted.

## C. Power and Energy Measurement Method

The energy that each encryption algorithm uses is the sum of the dynamic power and the static power integrated over the time that it takes to perform the encryption. For the hardware implementations we measure the dynamic power of both the sensor node and the FPGA. For the software implementations it is of course sufficient to only measure the dynamic power of the sensor node. The static power of each device is obtained from the manufactures' estimates since directly measuring static power was deemed difficult enough to place it outside the scope of this project.

TABLE V
STATIC POWER

| Device | Static Power (W) |
|---|---|
| Mica II | 0.003 |
| Spartan 3E | 0.008 |

The instantaneous dynamic power is recorded by attaching a current probe in series with the power supply of the device being tested. In the case of the FPGA we were only able to measure the core voltage because the other power supplies on the development board feed power to peripherals as well as to the FPGA. The signal from the current probe appears as a voltage waveform on an oscilloscope. The particular current probe we used has the relationship of 5mV/mA so we divided the voltage waveforms by 5 to obtain the actual instantaneous current. The waveform is then multiplied by the supply voltage (to obtain instantaneous power) and integrated over the length of time it took to perform the encryption to find the total energy.

This process is automated through the use of Matlab and a custom driver program for the that sets the correct oscilloscope modes and prepares the it to collect the data. The oscilloscope begins recording data when it encounters a rising edge on the trigger line (Channel 2). The trigger is explicitly set by the device being tested so that data is recorded exactly at the start of encryption. The device continues to set triggers at different points in the encryption cycle to aide analysis but the oscilloscope only responds to the first trigger edge. Once both the instantaneous current and the trigger data has been captured (see Figure 7), Matlab is used to find the energy.

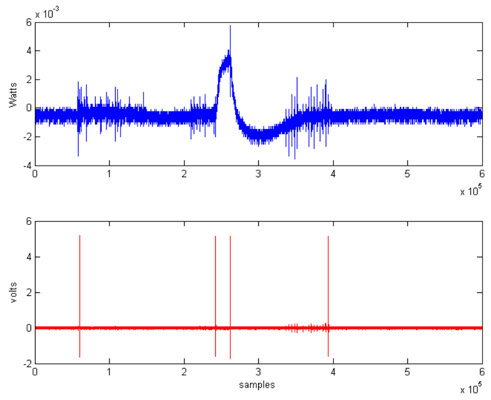

Fig. 7. Top: instantaneous FPGA power for encrypting one block of data. Bottom: Triggers showing start of data and key transmission, start of encryption, start of transmission of encrypted data, and finish point of encryption cycle.

The formulas used to compute the energy for each encryption cycle are:

$$E = (P_{avg})\,(T_{encrypt}) \tag{1}$$

$$E_{dyn} = \left(\frac{1}{N}\sum_{n=0}^{N}\frac{V_{probe}[n]}{5\Omega}V_{supply}\right)\left(\frac{N}{f_{samp}}\right) \tag{2}$$

Total energy for the hardware implementations is

$$E_{tot} = E_{dyn:fpga} + E_{dyn_mote} + E_{stat_fpga} + E_{stat_mote} \tag{3}$$

. The total energy for the software implementations is only

$$E_{tot} = E_{stat_mote} + E_{stat_mote} \tag{4}$$

. Note that we are ignoring all FPGA power in these calculations other than power supplied by Vcore. This will skew our results.

## IV. Results

### A. Algorithm Size

All the implementations were of comparable size with some notable differences. PRESENT in hardware has a much smaller encryption core than the other implementations but because of the way it reads the key in one clock cycle instead of multiple clock cycles, the key store was much larger and the total size was similar. PRESENT can be greatly reduced in size by modifying the way it reads the key.

TABLE VII
Size on Mica2 (total of 128 KB available)

| | | Bytes in ROM | |
|---|---|---|---|
| | Algorithm | No Optimizations | O3 |
| HW | AES | 1276 | 3588 |
| | TEA | 1992 | 2998 |
| | PRESENT | 766 | 1712 |
| SW | AES | 970 | 3956 |
| | TEA | 978 | 3220 |
| | PRESENT | 1148 | 3466 |

### B. Timing

Not surprisingly the hardware implementations are much faster than the software (Figure 8). The interesting result is how slow PRESENT is in software. It takes an order of magnitude longer than next slowest implementation. This is due to the large number of bit level manipulations performed by this algorithm. Swapping bits in hardware is no problem but in software many repeated shifting and masking operations are required. Also, in order to decide which bits should be swapped (this happens during the permutation layer), multiplication and division operations need to be performed which add even more time to total required for computation.

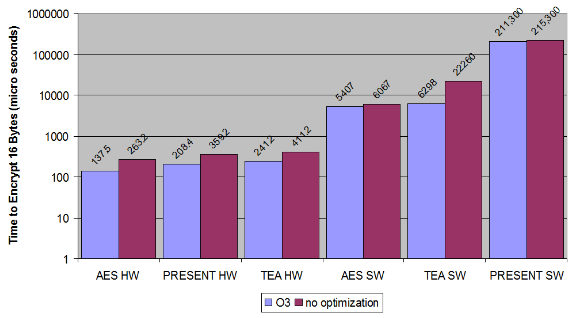

Fig. 8. Time required to encrypt 16 bytes.

Another interesting observation is the effect that compiler optimizations has on the speed of the encryption. Even when doing hardware encryption, optimizing the code resulted in very large time savings. Since total energy it directly dependent and how much time the encryption takes we choose to use O3 level optimizations for all tests.

TABLE VI
SIZE ON FPGA

| | FPGA Slices | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | Core | Key Store | Control | Other | Total | %on xc3s500e | Maximum Clock(MHz) |
| AES | 424 | 4 | 24 | 22 | 471 | 10 | 75.5 |
| TEA | 423 | 0 | 22 | 8 | 451 | 9 | 73.8 |
| PRESENT | 314 | 64 | 23 | 6 | 401 | 8 | 153.0 |

## C. *Energy Usage*

The total energy usage of each algorithm is shown in Figure 9. As expected the faster algorithms use less energy than the algorithms that take more time to finish. An interesting result is that hardware versions of TEA and PRESENT are less efficient than AES. Of the three algorithms AES is the largest and most complex so we would expect it to use the most energy. It turns out to be more efficient because of a problem with our implementations of TEA and PRESENT.

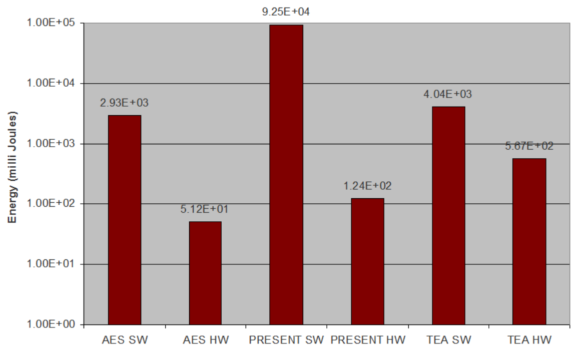

Fig. 9. Energy required to encrypt 16 bytes.

Both of these algorithms take in 8 bytes of data from the sensor node, encrypt the data, and then send it back. But since we are measuring for 16 bytes total the process has to be completed twice. This means that the port between the sensor node and FPGA has to be reversed a total of 3 times (only once for AES). This port reversal process is time consuming and adds significantly to the amount of time it takes to finish encrypting 16 bytes. Figure 10 dramatically illustrates how much energy is wasted on communication between the FPGA and sensor node.

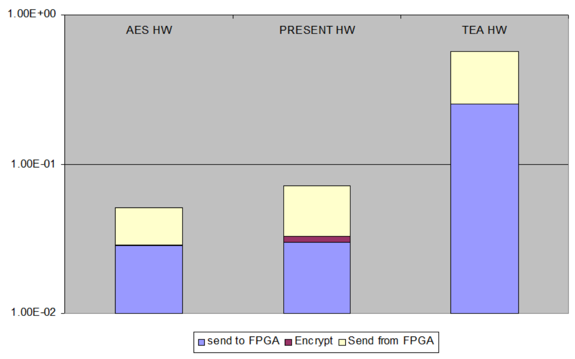

Fig. 10. Relative energy usage per operation mode.

*1) Reality Check:* The results in this section seem to indicate that there is no reason to every use encryption in software. If hardware is faster, more energy efficient, and more secure then why would anyone want to use a software solution? Well, in this project we were unable measure any of the FPGA supplies other than Vcore since the other supplies are used by peripherals on the development board. If these power supplies were taken into consideration then we suspect that the hardware solutions would actually be quite a bit less power efficient than the software solutions. This prediction is reasonably because one of the largest consumers of power is setting FPGA outputs. None of the core voltage goes directly to perform this task and as a result none of our power measurements include the energy used by the FPGA as it sends back the encrypted data to the sensor node.

## V. PROJECT ADMINISTRATION

### A. *Analysis of Success*

There are some parts of the project that we had to drop because of time that we would have liked to finish. Most notably we were not able to finish testing the PCB that the previous team designed. Had we finished, then we would have been able to get accurate measurements of all three FPGA power supplies.

We also had to cut the UART from the list of comparisons to make. This is not a total loss however because we did compare the speed of the two communication methods and determined that the new parallel interface is roughly 5 times faster than the UART (Figure **??**). In the previous section we already discussed how much energy is begin wasted on communication; it makes not sense to waste more by using a slow method of moving data between the sensor node and FPGA. The only scenario where someone would want to use a serial interface is if they could not spare the pins on microcontroller. We used 9 pins for our parallel interface which was 27%of the available pins on the Mica2.

We wish that we could have had time to examine many other facets of energy efficiency but feel like we got a good start. Our greatest success is that we implemented a reliable, extendable system that can be used to conduct further power measurements and can support other encryption algorithms. We knew that we would not be able to do everything for this project so we intentionally designed our testbed with modularity in mind.

### B. *Changes to Design*

We originally intended to create a fully universal framework for the top level FPGA design so that we could swap

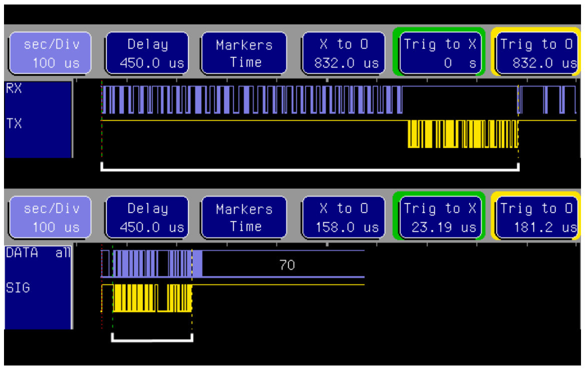

Fig. 11. Top: Encryption cycle via UART. Bottom: Encryption cycle via FPGA

encryption cores with ease. As we learned more about each encryption standard, we realized that such a framework would be very complex. Since the main focus of the project is to achieve maximum energy efficiency we decided to reduce the complexity of our design (and thus reduce the area) at the cost of some reusability.

### C. Surprises

We had no experience with multiple clock domains and failed to realize that we needed to sample the clock/control signal from the microcontroller multiple times on the FPGA to avoid setup and hold time violations. Rectifying this mistake caused delays as we had to repeat some of our design work.

Due to failure to properly read the datasheets throughway, we failed to realize that when both the microcontroller and the FPGA are in receive mode, the microcontroller would default to a logic 1 instead of 0 due to pull-up resistors [7]. The microcontroller pin for controlling the radio was also being used for signalling the FPGA, requiring us to reallocate it.

In addition, documentation for frequency settings to be placed in preprocessor defines for the radio was not updated for TinyOS 2.1, but was buried in TinyOS 1 documentation [8]. The radio does not seem to have flow control or reliability implemented on the network layer, requiring us to implement stop-and-wait to provide a crude degree of reliability.

Due to endianness issues, our test vector results for XTEA in hardware did not agree with XTEA in software. Also, the use of 32 bit integers with XTEA on the 8 bit microcontroller caused the microcontroller to malfunction, requiring a port of the XTEA algorithm to 8-bits.

### D. Remaining Issues

The AES hardware driver, and probably the rest of the hardware drivers do not reset properly after one encryption command. A few of the implementations have issues working with more than 16 blocks of 16-byte data at a time, most likely due to the use of 8 bit instead of 16 bit indices. The stop-and-wait reliability protocol is fairly inefficient, and changing it would also require adding a command packet type to start encryption and/or tell how many packets of data to be received. The large degree of packet loss without a reliability protocol needs to be investigated further.

In addition, the packets are not as large as they can be, decreasing efficiency [9] [10]. The drivers may need to be rewritten to support interrupts instead of polling for FPGA initialization, which has not been implemented due to the PCB not working yet. Finally, TEA has not yet been actually “TinyOSified” into small tasks. This was not an issue for our project as no radio packets are transmitted while encryption is occurring.

## VI. Future Work

### A. A Reliable Method of Measuring Energy

The power measurement method that we used is troubling because it is difficult to get accurate measurements for both parts of the system (FPGA,sensor node) at the same time. In addition it is difficult to interpret the data once it has been acquired because it has to be scaled and shifted by constants that are reported by the oscilloscope to the PC.

Some have suggested that using a large capacitor that has been fully charged as the sole power supply to the system is the best method for measuring how much energy is used by the system. After an operation has been finished the voltage on the capacitor can be measured and from that the total energy used by the system can be found. This method does have its draw backs because it does not give you any idea of the instantaneous energy usage. Future work would be to use the capacitor method to confirm the measurements taken with the oscilloscope.

### B. Other Communication Methods

Implementing SPI would be faster than the UART but still use fewer pins than the parallel interface that we designed. This would result in saved power (need to drive fewer pins) and use less microcontroller and FPGA resources. We would have like to have implemented SPI ourselves but we began discussing it too late to seriously consider it.

### C. Assembly

Writing the software implementations (or parts of them) in assembly to maximize their speed would allow for better comparison between the software and hardware implementations.

## VII. Conclusion

Though we did not finish everything that we have liked to have done, we did make a good start towards understanding the issues involved with adding encryption to lower-power embedded systems. Our system is well modularized and is ready to be extended by any one who would like to do so.

## Acknowledgment

We would like to extend our gratitude to our advisor Dr. Jens-Peter Kaps for his advice (and patience!) throughout the course of this project. Thanks also to Dr. Kris Gaj for his advice. Thanks to the previous team members: Brandon Thompson, Supreet Kaur, and James McCall for meeting with us to explain their work and give us a jump start into this project. Thanks also to Rajesh Velegalati for his help with the oscilloscope scripts. And finally thanks to all the members of CERG for their advice concerning our presentation and for bearing with us while we noisily conducted this project.

## References

[1] B. Thomson, J. McCall, and S. Kaur, "Cryptographic coprocessor for wireless sensors," George Mason University, Fairfax, VA, USA, Senior Design Project Report, Dec. 2008.
[2] "Tinyos documentation wiki." [Online]. Available: http://docs.tinyos.net/index.php/Main_Page
[3] I. O. Levin, "A byte-oriented aes-256 implementation," 2007. [Online]. Available: http://www.literatecode.com/2007/11/11/aes256/
[4] D. Wheeler and R. Needham, "Standard c implementation of xtea." [Online]. Available: http://en.wikipedia.org/wiki/XTEA
[5] "C present implementation (8 bit)," 2007. [Online]. Available: http://www.lightweightcrypto.org/present/
[6] "Interfaces and components." [Online]. Available: http://www.tinyos.net/tinyos-2.x/doc/nesdoc/mica2/
[7] A. Inc., "8-bit AVR microcontroller with 128K bytes In-System programmable flash - ATMega128/ATMega128L." [Online]. Available: http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf
[8] "CC1000 radio stack manual," http://www.tinyos.net/tinyos-1.x/doc/mica2radio/CC1000.html, 2003. [Online]. Available: http://www.tinyos.net/tinyos-1.x/doc/mica2radio/CC1000.html
[9] P. Levis, "message_t." [Online]. Available: http://www.tinyos.net/tinyos-2.x/doc/html/tep111.html
[10] J. A. Sobrino, "[Tinyos-help] maximum packet size." [Online]. Available: https://www.millennium.berkeley.edu/pipermail/tinyos-help/2009-February/038495.html