AES Hardware-Software Co-Design in WSN

Carlos Tadeo Ortega Otero, Jonathan Tse, and Rajit Manohar
Computer Systems Laboratory
Cornell University
Ithaca, NY, U.S.A.
{cto3,jon,rajit}@csl.cornell.edu

***Abstract*—Wireless Sensor Networks (WSNs) present a challenging design space for encryption algorithms. We evaluate hardware, software, and hybrid implementations, including one of our own design, of Advanced Encryption Standard (AES) encryption engines in the context of WSN microcontrollers. We examine the tradeoffs between energy, throughput, memory footprint, and sensor network node lifetime. Our measured results and models show that our fully Quasi Delay-Insensitive, asynchronous AES design, combined with a low-power microcontroller, offers a 60× increase in throughput at 90× *less* energy per bit over the commercially available TI MSP430 AES WSN hardware. Our hardware AES offers a 30× throughput improvement over its software counterpart, albeit with reduced lifetime. By incorporating power gating and providing dedicated memory resources to the AES engine, hybrid implementations can provide a 6× better throughput and increase the lifetime by 10 % over software.**

I. Introduction

Wireless Sensor Networks (WSNs) are becoming more prevalent in a myriad of applications ranging from medical monitoring devices to industrial control systems. Typical WSNs are comprised of many small, low cost nodes or *motes* that gather, process, and propagate data about their surrounding environment. Mote deployment lifetime is often measured in years, so minimizing mote energy consumption and static power is of great interest.

Aggressive energy reduction at the cost of performance is a common engineering tradeoff for mote microcontrollers. The Atmel 128L-based MICA2 [6] mote is an example of a relatively high-performance, high-energy point in the tradeoff space. Smartdust [23] is low-energy (12 pJ per instruction) and low-performance (500 kHz). The Phoenix microcontroller is an extreme example, offering 106 kHz at 2.8 pJ per instruction [24]. Recent application of asynchronous circuit techniques has improved the energy-performance tradeoff—ULSNAP is a low-energy, fully Quasi Delay-Insensitive (QDI) asynchronous mote microcontroller design that offers 93 MHz of performance at only 47 pJ per instruction [21].

High-performance mote microcontrollers like ULSNAP and MICA2 allow more complex computations locally at the mote, which in turn can reduce the duty cycle of the energy-hungry wireless communication systems. However, we must also account for the energy consumption of encryption, as transmitted information is oftentimes sensitive and should be encrypted to preserve confidentiality [20]. As of this writing, the *Advanced Encryption Standard (AES)* [17] is approved for both non-classified and classified information by the US Government and has has become the industry standard for encryption in applications ranging from SSL to storage media encryption.

Implementing AES in the WSN space is complicated by the need for a small energy envelope. Typical activity patterns in the WSN application space are "bursty," i.e. long quiescent periods followed by a brief, highly-active period. Asynchronous circuits are an excellent fit for these activity patterns as they are event-driven, i.e. completely idle in the absence of data to compute. This intrinsic clock-gating-like behavior of idle asynchronous circuits eliminates switching energy during these idle periods without sacrificing performance.

As energy consumption is a metric of interest for WSNs, we have developed a model for mote battery lifetime in Section II, which accounts for both the energy of encryption and idle power. To aid the WSN architect, we present software, hardware, and hybrid implementations of AES encryption on WSN microcontrollers, using our model to evaluate the impact to sensor mote lifetime of each class of implementation. Our hardware and hybrid implementations are asynchronous to take full advantage of the aforementioned energy benefits. Our WSN mote evaluation platforms are the MSP430 (CC430F6137) [7] and ULSNAP [21], representing the state-of-the-art in WSN platforms providing more than 10 MIPS of performance in industry and academia, respectively.

II. Mote Lifetime Model

We adapt the model for sensor node lifetime analysis used in the work of Jung et al. [8] to include the effects AES encryption on battery lifetime. The model is a semi-Markov chain formulation of the power state transitions and satisfies the following properties: ergodicity and Poisson distribution of event arrivals. An event arrival could be the availability of new sensor data, a timer expiring and forcing the mote to execute a pre-planned action, the arrival of a command via the radio, etc. We assume the processing, and transmission times are independent and identically distributed with an arbitrary distribution.

Figure 1 shows our model for the power states of a generic cryptographic WSN mote: Sensing (S_1), Processing (S_2) and Transmit or TX (S_3). The TX state S_3 is an embedded chain of an encryption state, S_E, and a data transmission state, S_T. The processing and communication steps are optimized for throughput and do not enter a low-energy/low-performance mode when work is available. α represents the probability of data transmission after processing. Figure 2 is a sample encryption power trace. $\overline{\mu_1}$, $\overline{\mu_2}$, and $\overline{\mu_3}$ are the expected averages of *sensing time*, *processing time*, and *communication time*, respectively. The event inter-arrival time is $1/\lambda$. We assume that motes

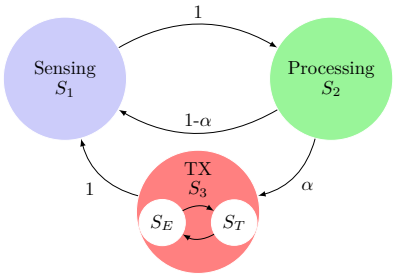


Fig. 1: Semi-Markov Chain for WSN Mote

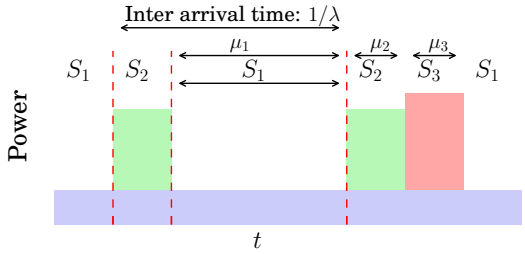


Fig. 2: Mote Power Profile

rarely receive commands or signals, i.e. they are primarily collecting, processing, and transmitting sensor data. In a case where WSN designers expect significant receive traffic, our model could be extended to include a fourth Receive or RX state. We do not include an RX state here, as we focus primarily on evaluating our AES *encryption* engine, as described in Section III.

For both ULSNAP and the MSP430, state S_1 is the idle state—i.e. it has the lowest activity factor of all states. In state S_1, ULSNAP is effectively clock-gated due to its QDI circuit implementation [21] and the MSP430 enters a low-power state. The static power consumption of both systems is a reasonable proxy for the power consumption in S_1.

Each state S_i consumes $E_i = t_i P_i$ energy, calculated from the consumption P_i and the total time spent in a state t_i. Of course, $E_{\text{total}} \geq \sum_i E_i$. Over a long period, the total time spent at state S_i is given by $t_i = \lim_{t\to\infty} p_i t$, where p_i is the proportion of time spent in state S_i. Therefore, $E_i = p_i t P_i$. Let π_j be the *stationary probability* of the Markov chain, which is the frequency of visiting each state over an infinite execution. p_{ij} is the probability of transitioning from S_i to S_j. π_j can be interpreted as the proportion of transitions into S_j.

$$\pi_j = \sum_i \pi_i p_{ij}, \quad \sum_j \pi_j = 1 \tag{1}$$

The probabilities p_{ij} can be obtained from the Markov chain in Figure 1. The equations can be written in matrix form, where row indices represent source states and column indices represent end states:

$$[\pi_1 \quad \pi_2 \quad \pi_3] \begin{bmatrix} 0 & 1 & 0 \\ 1-\alpha & 0 & \alpha \\ 1 & 0 & 0 \end{bmatrix} = [\pi_1 \quad \pi_2 \quad \pi_3] \tag{2}$$

The Markov chain model allows us to express p_i as follows, where μ_i is the mean time spent in S_i before making a transition—$\overline{\mu_1}$, $\overline{\mu_2}$, and $\overline{\mu_3}$, respectively:

$$p_i = \frac{\pi_i \mu_i}{\sum_j \pi_j \mu_j} \tag{3}$$

Equation (2) and (3) allow us to re-write E_{total} and extract t as the mote lifetime:

$$E_{\text{total}} \geq t \frac{\overline{\mu_1} P_1 + \overline{\mu_2} P_2 + \alpha \overline{\mu_3} P_3}{\overline{\mu_1} + \overline{\mu_2} + \alpha \overline{\mu_3}} \tag{4}$$

$$t_{\text{life}} \leq \frac{E_{\text{total}}(\overline{\mu_1} + \overline{\mu_2} + \alpha \overline{\mu_3})}{\overline{\mu_1} P_1 + \overline{\mu_2} P_2 + \alpha \overline{\mu_3} P_3} \tag{5}$$

Assuming that $\overline{\mu_1} \gg \overline{\mu_2}, \overline{\mu_3}$, we approximate the average sensing time $\overline{\mu_1}$ as the inter-arrival time $1/\lambda$. We also expand the power state S_3 into its embedded Markov chain of S_E and S_T, which means that the mean time spent in S_3 can be expressed as the sum of the mean times spent in S_E and S_T, i.e. $\overline{\mu_3} = \overline{T_E} + \overline{T_T}$. We can rewrite (5) as:

$$t_{\text{life}}(\lambda) \leq \frac{E_{\text{total}}(1 + \lambda(\overline{\mu_2} + \alpha(\overline{T_T} + \overline{T_E}))}{P_1 + \lambda(\overline{\mu_2} P_2 + \alpha(\overline{T_T} P_T + \overline{T_E} P_E))} \tag{6}$$

We use (6) in Section IV to compute mote battery lifetime, a key figure of merit for our evaluation.

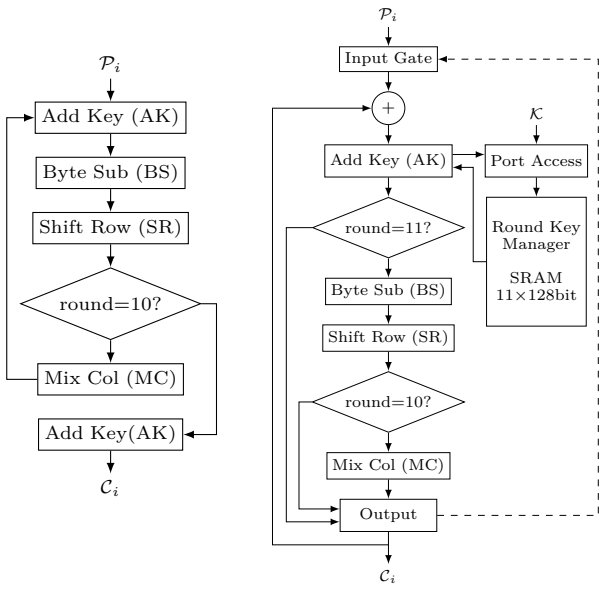
III. AES Implementation

AES is a 128-bit block cipher with 128-, 192-, or 256-bit keys ($\mathcal{K}$). Typically, the blocks are logically organized into a 4×4 *state* matrix of 8-bit elements. A full encryption/decryption step is comprised of several loop iterations or *rounds*—10, 12, or 14 rounds for 128-, 192-, and 256-bit keys, respectively. Each round does four operations on the state matrix [17]:

1) Add Key (AK)
2) Byte Substitution (BS)
3) Shift Rows (SR)
4) Mix Columns (MC)

Figure 3a shows a simplified dataflow diagram for a complete execution of a 128-bit key AES encryption. The contents of Figure 3a are then wrapped with the appropriate control structures to implement the desired cipher mode of operation. There are several choices as to which cipher-block mode to implement. We chose Cipher-Block Chaining (CBC) as it is widely used in the WSN context [9] and the simpler Electronic Code Book (ECB) is vulnerable to several attacks [18].

Many WSN motes follow a single-threaded execution model to reduce energy consumption. Thus, for a software implementation, the cost of ciphering a single block governs the overall encryption performance for all modes of operation. There is also the concern of limited memory space for programs on embedded systems. In CBC mode, common hardware optimizations such as pipelining are unattractive in the context of a single encryption because of the carried loop dependencies. In practice, some degree of pipelining is still desired to logically separate the various



(a) AES Block Cipher (b) CBC Implementation

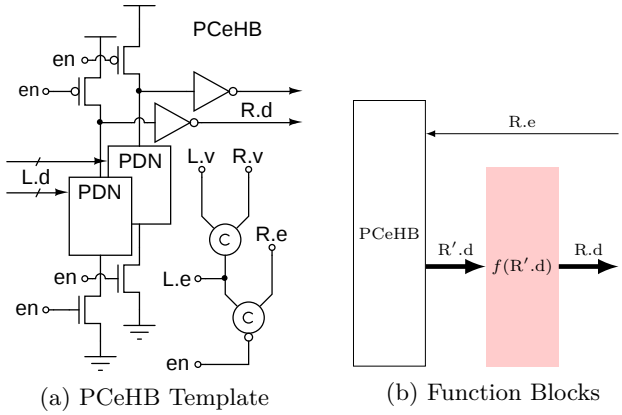
Fig. 3: AES Implementations

stages of encryption, reduce signal fanout, and ease system design.

While WSN motes can occasionally receive encrypted commands, we assume this is very infrequent and that the payload is small. To support decryption, we assume the existence of a lightweight AES decryption engine in software. This assumption allowed us to make a number of optimizations to our hardware AES encryption engine, as we did not have to support decryption. We note the optimizations as well as the requirements to support hardware decryption below where applicable.

Figure 3b illustrates our encryption engine hardware implementation. The unrolled encryption key is stored in an externally user-writeable SRAM. However, only the *AK* process can read the SRAM. SRAM writes and reads are arbitrated by the *Port Access* process. While our AES engine is implemented at the circuit level as a pipelined system—each box represents a pipeline stage, we treat the system as an un-pipelined functional unit when doing a single encryption. The *Input Gate* process blocks further inputs until it receives a "done" signal from the *Output* process. Although the pipelining of a single message encryption is unattractive due to data dependencies, we can take advantage of the circuit-level pipelining of our implementation by operating on multiple messages. A small change to the *Input Gate* and *Output* processes allows us to pipeline encryption of multiple messages by interleaving the plaintext and ciphertext blocks of each message as appropriate.

Our AES implementation (AES-QDI) makes use of the Quasi Delay-Insensitive (QDI) family of asynchronous circuits. We use Martin Synthesis [12], which breaks apart a computation into fine-grained hardware processes that communicate over point-to-point delay insensitive channels. Instead of using a clock and flip-flops for synchronization and storage, QDI circuits use channel handshakes for local, inter-process synchronization and represent data as tokens traversing these channels. In fact, our AES state matrix is implemented as in-flight data tokens as opposed



(a) PCeHB Template (b) Function Blocks

Fig. 4: PCeHB Implementations

to being stored in a set of registers. The initial state matrix, i.e. the matrix before the AK step, is input to the encryption engine as a token stream. Once the encryption is complete, the ciphertext is obtained by collecting the output token stream. The overall structure of the state matrix is implemented by proper connectivity and control in the mux/demux stages between each AES encryption step.

QDI hardware processes and the QDI channels themselves are robust to arbitrary gate delays. As a result, QDI circuits are intrinsically tolerant to process, temperature, and voltage variations. QDI circuits are also naturally event-driven, waiting in a quiescent state with no switching activity until a data token arrives. This is the equivalent of perfect clock-gating in a synchronous system—inactive processes consume only leakage current. Since encryption is typically only active during data transmission in the WSN application space, a QDI implementation effectively offers instantaneous transitions between wake-up/sleep states with perfect clock-gating.

Our AES-QDI implementation was synthesized with PreCharge Half/Full Buffer pipeline templates (PCeHB/PCeFB) [3,11], which have been widely used in multiple QDI designs [14]. Each PCeHB/PCeFB stage implements a function of simple enough complexity that it can fit into a single nMOS Pull-Down Network, as shown in Figure 4a. This compilation style yields high-performance stages with short cycle time and only 2 forward transitions.

The signals L.v and R.v are validity signals calculated by inspecting the input and output data rails L.d and R.d, respectively. As the data rails are encoded with a delay-insensitive protocol, we can obtain validity by inspecting voltage independent of time. We primarily use the half-buffer PCeHB templates for computational pipelines. While these templates are high-performance, implementing reasonably complex functions usually results in a deep pipeline of PCeHB stages.

As discussed earlier, the carried loop dependencies inherent in to CBC AES discourage fine-grained pipelining as it increases overall latency. To avoid an over-pipelined design, we combined PCeHB templates with asynchronous function blocks [13], which can be thought of as the asynchronous analogue to combinational logic in a clocked pipeline stage. The between-stage handshaking is still handled by the PCeHB handshaking logic, but the computation is broken up across the PDN and additional

function blocks placed in series, as seen in Figure 4b. Validity is calculated after the functional blocks have finished computation, with the PDN effectively implementing the first function in the series chain. This reduces the handshaking overheads and forward latency per pipeline stage, reducing the overall latency.

To evaluate our design, we performed detailed transistor-level simulations in SPICE, complete with parasitics and conservative wire-loads. Our asynchronous design flow does not use a traditional ASIC standard cell flow. We typically generate cells on-demand using CAD tools such as *cellTK* [10], which are then place and routed. Thus, a simulation of a transistor-level SPICE netlist as described represents a simulation of a completed digital design. We compare our implementation with state-of-the-art AES implementations in Section IV-B. We describe the individual components of the AES system below:

A) Add Key (AK): AK performs a bitwise XOR of each byte in the state matrix and the corresponding byte of the current input block. We implemented this block using a PCeHB template with the XOR encoded in the PDN, as the complexity of an XOR does not warrant additional asynchronous functional blocks.

B) Byte Substitution (BS): Each byte of the state matrix transformed by a non-reversible, non-linear function. In the case of AES, we use a $GF(2^8)$ Galois Field. Designers typically focus their attention on the BS block as it uses almost 75 % of the energy required for an encryption. There are two common ways to implement the BS block: 1) Composite Field Transformation (CFT) or 2) Look-Up Table (LUT) [4,15].

CFT is oftentimes implemented as multiplicative inversion followed by an affine transformation. In order to minimize area and pipeline the circuit, the multiplicative inversion is performed in an equivalent Galois Field, e.g. $GF((2^4)^2)$. This method usually yields small, pipelined circuits with high propagation delays. The composite field transformation can be used for both encryption and decryption with minimal changes. CFT-based blocks usually have high throughput, but the long-forward propagation delay makes CFT unattractive for many AES implementations. However, if overall silicon area is limited, as is the case in some WSN motes, CFT remains a good option.

A LUT encodes the output values for all possible 8-bit input combinations. Possible ways to build this lookup table include using a Product-of-Sums or Sum-of-Products (SOP), or by simply implementing a ROM-based LUT. More advanced techniques include the use of Binary Decision Diagrams or Twisted Binary Decision Diagrams (TBDD) [15]. Automated synthesis using commercial tools typically results in the SOP implementation. A ROM implementation typically has lower propagation delays than the SOP, but requires a special ROM compiler and double memory to support both encryption and decryption.

Satoh et al. show that the propagation delay of a BS stage using CFT is 2190 ps in a 130 nm technology. In contrast, a table lookup takes 700 ps, but at the cost of a 4.5x increase on the number of transistors [15]. Satoh et al. also propose a TBDD solution that reduces the delay to 430 ps, with a 8x increase in area compared to the composite field version.

In our 90 nm technology, we implemented both a CFT and a ROM-based LUT implementation of the BS function. Our CFT implementation has a pipeline depth of 5 stages, as shown in Figure 5. The *prologue* transforms the 8-bit vector from a $GF(2^8)$ to a $GF(2^4)^2$. The inversion is performed in this Galois Field and then the *epilogue* transforms the resulting 2 nibbles to the $GF(2^8)$. Each stage of the CFT-based BS is implemented using PCeHB buffers combined with asynchronous functional blocks, as described earlier and as shown in Figure 4b. We simplified the logic using EXORCISM, a minimization tool for of Exclusive-Sum-Of-Products (ESOP) expressions [25].

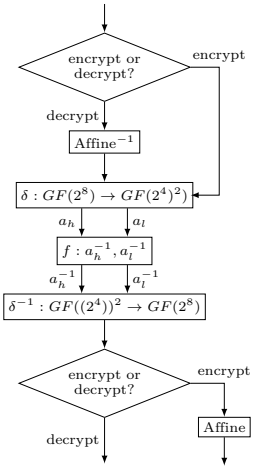


Fig. 5: Composite Field Transformation Byte Substitution

Our ROM implementation is a 256-row, 8-bit wide NOR-based ROM array. The architecture of our ROM is similar to that described by Nystrom et al. [19]. A ROM access has four stages: 1) receive input, 2) decode input to address, 3) access ROM at address, 4) transmit output. While our ROM implementation has 23 % less throughput than CFT, it has 35 % less propagation delay. As the AES CBC algorithm is constrained by loop data dependencies, minimizing delay is a key contributor to performance. Thus, we used the ROM-based BS for AES design, discussed in Table IV.

C) Shift Rows (SR): SR implements a cyclic left shift for encryption on each row of the state matrix. The shift amount for the n^{th} row is n bytes, indexing from zero. If supporting both encryption *and* decryption is required, i.e. both left and right shifts are necessary, we can compile SR using PCeHB templates to implement the variable direction shift. Our AES engine is intended for encryption so we simply implement the shifting with wires, reducing the delay and energy to the cost of transmission on wires.

D) Mix Columns (MC): MC applies a column-wise 32-bit linear transform to the state matrix. Input columns, denoted a, are treated as a polynomial over $GF(2^8)$. They are then multiplied by the fixed polynomial $c(x) = 3x^3 + x^2 + x + 2$ modulo $x^4 + 1$ to obtain the output column, denoted b. This is equivalent to the matrix vector multiplication shown in (7).

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{7}$$

Solving for elements of b requires addition operations as well as multiplication by factors of 2 and 3. Addition over $GF(2^8)$ is an XOR and multiplication by 2 is done via shifting followed by an XOR with a constant. Multiplication by 3 can be achieved by a multiplication by two and an XOR addition. Our MC is architecturally similar to [1] and is implemented with PCeHB templates and 2 cascaded function blocks, as shown in Figure 6. Implementing MC for decryption requires multiplication by 9, 11, 13, and 14. These multiplications are typically implemented as lookup tables. Implementing only encryption allows us to avoid the additional overheads of lookup table ROMs.

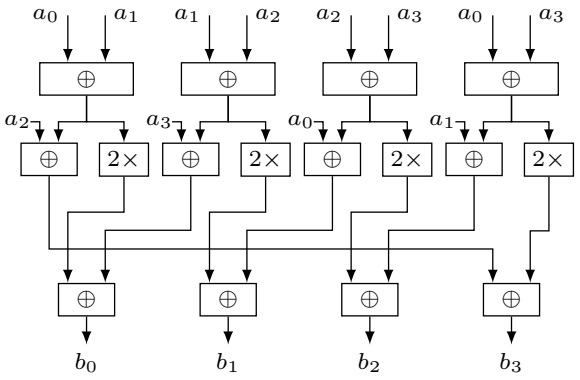


Fig. 6: Mix Columns

IV. AES Evaluation

In our study, we benchmarked and analyzed software, hardware, and hybrid AES implementations for two microcontrollers: the commercially available TI MSP430, version CC430 [7], and ULSNAP [21]. A similar analysis can be applied to any other microcontroller platform. The MSP430 provides a completely in-silicon solution for AES, which serves as a data point for commercial WSN AES hardware. ULSNAP is a microcontroller design from academia targeted for the WSN space. ULSNAP employs a number of advanced techniques at both the circuit and microarchitectural levels to improve energy efficiency while maintaining performance [21]. The MSP430 CPU implements a 16-bit, single-pipeline Von Neumann architecture with a modern RISC ISA. ULSNAP is also 16-bit and has a MIPS-like RISC ISA, but implements a Harvard architecture. While ULSNAP does not have explicit active power modes, by changing the operating voltage the user can choose high-performance and low-power characteristics.

ULSNAP makes use of the QDI family of asynchronous circuits [21]. That makes ULSNAP robust to systemic and random variations from manufacturing, temperature and voltage. The compilation of QDI processes implements the equivalent of the perfect clock-gated synchronous circuits– inactive processes consume only leakage current. With respect to our model described in Section II, we assume regularly spaced sensor events, each of which has probability α of causing the transmission of a single encrypted packet after being processed.

We have access to a number of packaged, functional 90 nm ULSNAP dies. All ULSNAP software implementation results are measured data from actual on-silicon executions of compiled code. The TI CC430 is commercially available, so all results which do not involve our AES implementations are from measurements of a physical MSP430 device. Results for our AES implementation are from SPICE simulations of a completed, sized, transistor-level design netlist annotated with conservative wire and other parasitic capacitances.

As described in Section III, our transistor-level netlist is a complete design, ready for place and route and subsequent fabrication. Our SPICE simulation flow is well-characterized against silicon chips and often returns more conservative numbers than simulations of extracted layout and actual die measurements. Results for the hybrid configurations below are a combination of SPICE simulations and measurements from silicon as appropriate.

The total size of a transmitted packet packet is 133 B with a payload of 127 B. This payload is the maximum allowable for a Zigbee packet as defined by the IEEE 802.15.4 standard. The transmission time of a packet, T_T, is set by the transmission rate. We assume that our motes use the TI-CC1101 transceiver, which offers 500kbps of bandwidth at 55 mW [26]. Using this transceiver is a natural fit for the MSP430, and we assume ULSNAP can be paired with the CC1101 or an equivalent radio. Transmitting a 133 B packet takes $T_T = 2.12$ ms.

We assume the average processing time for each event to be $\overline{\mu_2} = 1.1$ ms, which is the average completion time for the statistical benchmark set from the SenseBench suite [16] running on ULSNAP [21]. $\overline{\mu_1}$ and $\overline{T_E}$ are dependent on the event inter-arrival time and AES implementation, respectively. Note that encrypting the 127 B payload takes 8 full encryptions, which we have accounted for in our evaluation below.

A. Software-Only Approach

In order to maximize energy efficiency, programmers can leverage different processor power modes. The MSP430 implements four active power modes, which offer a tradeoff between power and performance, and five sleep modes, which offer a tradeoff between static power and wake up time. In high performance (HP) active mode, the MSP430 core consumes 4.5 mA at 2.4 V, which translates to about 54 nJ per operation. For the MSP430, an operation equates to a complete instruction execution. When running in low-energy (LE) active mode, the MSP430 consumes 160 µA/MHz with a *minimum* voltage of 1.8 V. The maximum frequency in low-energy mode is 1 MHz for an energy per instruction of 28 pJ.

ULSNAP does not have discrete power modes but rather scales performance continuously with operating voltage due to its QDI circuit implementation [21] and enters sleep mode automatically once all events have been processed. At 1.2 V or HP mode, ULSNAP runs at 93 MHz and draws 3.45 mA—47 pJ per instruction. At 0.95 V or LE mode, ULSNAP offers 47 MHz at 1.3 mA, or 29 pJ per instruction on average [21].

Table I shows the power and performance for four different software implementations of AES running at the high and low power modes of our test microprocessors. TI-C and ULSNAP-C are evaluations of the same AES library written in C compiled for the MSP430 and ULSNAP, respectively. TI-MSP430 is a Texas Instruments software implementation of AES optimized for the MSP430.

TABLE I: AES Software Implementations

Design	Mode	Perf. [Mbps]	Power [mW]	Energy [nJ/bit]	Memory [B]
TI-C	HP	0.102	10.8	105.4	3441
	LE	0.005	0.2	56.1	
TI-MSP430	HP	0.420	10.8	25.3	1184
	LE	0.021	0.2	13.5	
ULSNAP-C	HP	1.550	4.1	2.6	2670
	LE	0.786	1.2	1.5	
ULSNAP-O	HP	1.850	4.1	2.2	2664
	LE	0.935	1.2	1.3	

TABLE II: AES Hardware Implementations

Design	Process [nm]	Perf.		Latency		Energy [pJ/bit]
[1,2]	130	141	Mbps	910	ns	79
[27] @ 1.5V	130	23	Mbps	5.6	µs	81
[27] @ 1.0V	130	8.1	Mbps	15.4	µs	43
[4] @ 3.30V	350	9.9	Mbps	12.9	µs	–
[4] @ 0.65V	350	12.8	kbps	10.3	ms	55000
[28]	180	1.6	Gbps	80	ns	300
[15]	130	10.0	Gbps	11.3	ns	191
MSP430 [7]	–	15.0	Mbps	8.5	µs	717
AES-QDI(ours)	180	907	Mbps	138	ns	34
AES-QDI(ours)	90	948	Mbps	135	ns	8

ULSNAP-O is an optimized software implementation of AES written by the designers of ULSNAP.

In general, software implementations on ULSNAP perform better than their MSP430 counterparts. For instance, the optimized TI library provides a maximum throughput of 0.42 Mbps. The ULSNAP core *quadruples* the MSP430 performance to 1.85 Mbps while using 10x *less* energy. On the other hand, the ULSNAP microcontroller uses much more memory than the TI implementations. This is mostly due to the differences in memory architecture—ULSNAP is word-aligned (16-bits) while the MSP430 allows for byte-aligned memory access.

B. Hardware-Only Approach

We define a "hardware approach" as any implementation of AES as a specialized *ASIC* or *transistor-level* coprocessor. The plaintext and unrolled key are transferred to this coprocessor and the ciphertext is returned, oftentimes freeing the main processor to engage in another task.

We augmented ULSNAP with an AES transistor-level implementation optimized for minimizing energy. The results in Table II come from transistor-level SPICE simulations which include wiring capacitance. We implemented our AES system in an 180 nm high-performance process and an 90 nm *low-power* process to match ULSNAP [21]. The static power consumption of ULSNAP-AES is 7.54 µW and 17.3 µW for the 180 nm and 90 nm versions, respectively. The low-power 90 nm process allows a similar performance to the 180 nm version at a reduced energy.

While we cannot compare all AES implementations, we show the reported numbers for the best-in-class implementations in Table II and Table IV. Figure 7 shows a comparison of state-of-the-art implementations on the energy-throughput space. Our implementation and the one implemented by Satoh [15] are in the energy/throughput Pareto optimal set of these implementations. It is worth noting that the implementations in [1,2,27] are also QDI and thus benefit from robustness to delay and perfect clock-gating-like behavior. While [1,2] share similar architecture and circuits with our implementation, our design is faster and more energy efficient. We attribute this

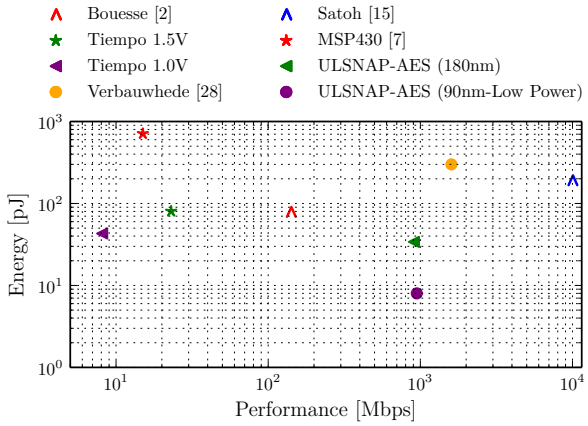


Fig. 7: AES Hardware Energy-Throughput Space

TABLE III: ULSNAP Software AES Blocks

Block	Mode	Delay [µs]	Energy [pJ/bit]	Memory [B]
AK	HP	3.30	106.82	452
	LE	6.54	63.05	
SUB	HP	1.25	40.55	1058
	LE	2.48	23.94	
MC	HP	3.88	125.62	552
	LE	7.68	74.15	
MEM	HP	2.14	69.24	40
	LE	4.24	40.87	

primarily to: 1) the low latency of our BS block, 2) our non-standard-cell-based synthesis method, 3) transistor sizing for performance, 4) our SRAM-based round key manager. On the other hand [1,2] claim balanced layout paths resistant to differential power analysis attacks, but no assurance of the safety of the device is given. Their design is also based on a commercial IP library, hence it can be optimized further in the Energy-Performance space. Our numbers for their design are from the instance described in their datasheet [27].

Table II also includes the MSP430's hardware implementation of AES. Our measurements indicate that the Processing step (S_2) takes 170 clock cycles, delivering 15.05 Mbps. However, an extra 140 clock cycles are required to load the plaintext into the encryption unit, set the coprocessor interrupts, and retrieve the data from the AES coprocessor, so the *net* performance is 8.5 Mbps.

C. Hybrid Approach

We partitioned the AES system into four parts, each of which can be implemented in either hardware of software: Loop Control (Ctrl), Add Key (AK), Byte Substitution and Shift Rows (BS), and Mix Columns (MC). We combined Byte Substitution and Shift Rows as a logical step as they are always executed sequentially, as seen in Figure 3b.

Table V enumerates all possible hybrid configurations. Note that Ctrl is only implemented in hardware if AK, BS, and MC are also hardware. For all software blocks we used the same AES library as used earlier. All of the ULSNAP configurations are evaluated in the 90 nm low-power process discussed earlier. Per-block measurements for both software and hardware ULSNAP implementations are available in Table III and Table IV, respectively. As we do not have detailed access to the netlist of the MSP430's AES implementation, we substitute our AES hardware implementation where appropriate for the following analysis of the MSP430.

TABLE IV: Hardware AES-QDI Blocks

Block	Perf. [MHz]	Energy [pJ/bit]	Txr [×1000]	Delay [ns]	P_{static} [µW]
AK	370	0.140	28.65	0.21	0.47
MC	281	0.136	14.57	0.57	0.80
BS - ROM	155	0.351	7.45	3.54	0.84
BS - CFT	192	0.750	5.49	4.83	†0.29
Ctrl/Key SRAM			107.7		11.30

†: Encryption and Decryption

We augment our model from Section II by defining T_{hyb} as the block encryption time of our hybrid system. T_{load}, as seen in (8), is the time to load data into the accelerator, which we assume is similar to the cost of accessing memory. T_{AK}, T_{BS}, and T_{MC} are the execution times for the AK, BS, and MC units, respectively. The coefficients below represent the total number of executions of each unit for a complete encryption, including the load time T_{load}:

$$T_{\text{hyb}} = 10T_{\text{load}} + 11T_{AK} + 10T_{BS} + 9T_{MC} \quad (8)$$

$$T_{\text{hw}} = \quad T_{\text{load}} + T_E \quad (9)$$

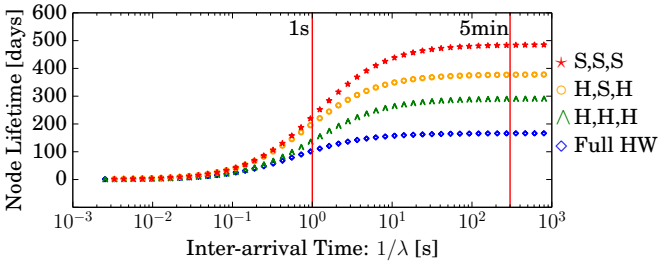
$$T_{HSH} = 10T_{\text{load}} + T_{\text{hyb}} \quad (10)$$

The differences between the hybrid execution time in (8) and the hardware execution time in (9) are as follows: A full hardware implementation needs a single access to the encryption data whereas a hybrid implementation must effectively access ten times by transferring data between the hardware and software blocks. T_E accounts for the double encryption necessary to send a 29 B packet. If AK and MC are implemented in hardware but BS is implemented in software, we incur an *additional* ten memory accesses as we need to retrieve the data twice, as shown in (10). We refer to this combination as "hardware-software-hardware (HSH)."

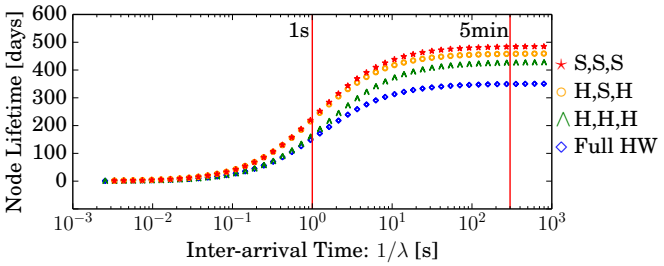
Table V lists our estimated performance and lifetime numbers for hybrid implementations of AES on ULSNAP and the MSP430, using our hybrid AES implementations alongside both processors. The throughput was calculated by adding the delay of blocks from Table III and Table IV as appropriate. As a validation of our estimates, the software only approach in Table V, configuration 8, matches the measured result in Table I (ULSNAP-C) to within 1 %. Similarly, the software only approach of the MSP430 matches the (TI-C) result from Table I. The TI-C software implementation fuses the AK and BS steps, so we did not evaluate hybrid combinations of AK and BS on the MSP430.

t_{life} in Table V is an estimate of mote battery life from our model, assuming an inter-arrival rate of $1/\lambda = 1$ min. The static power consumed in state S_1 accounts for that of the microcontroller as well as that of any hardware-implemented blocks, as estimated in Table IV.

Our full-hardware implementation is roughly 180k transistors, effectively 30 % of the reported 592k transistors comprising ULSNAP [21]. We assume that the BS computation is parallelized and requires four S-box. The Ctrl unit is roughly double the size of the sum total of all individual blocks. Table V shows an all-hardware implementation offers the best throughput. Moving to software-based control of the hardware results in a 10x penalty to throughput. However, a full-hardware implementation also has the *lowest* mote lifetime as the static power consumption from the additional transistors dominates due to the long idle time.



(a) Lifetime



(b) Lifetime with Power Gating

Fig. 8: ULSNAP Lifetime

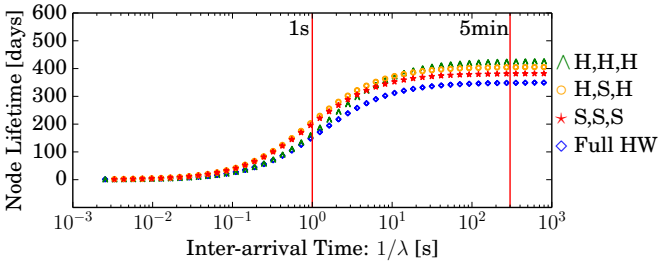


Fig. 9: ULSNAP Lifetime with Memory Overheads

Figure 8a shows the lifetime of various hybrid AES configurations alongside ULSNAP as function of the event inter-arrival time ($1/\lambda$). For $1/\lambda < 4$ s, the hybrid schemes offer better performance than a software-only implementation with little or no impact in the lifetime of the mote. In contrast, for sparse events t_{life} is governed by the static power. For inter-arrival times greater than 5 minutes, Figure 8a shows a gap in excess of 3x between the hybrid and software counterparts.

The simple addition of power-gating cut-off transistors in QDI circuits reduces idle power consumption. We sized the sleep transistors to reduce static power consumption by an average of 80 % with an average of 20 % performance degradation [22]. This is particularly beneficial for the full-hardware configuration. Incorporating these estimated power gating savings provides 50- to 100-percent improvement in mote lifetime for sparse arrival times, as seen in Figure 8b.

Figure 8a and Figure 8b assume that the software portions of the hybrid AES implementations share memory with the host microprocessor. Adding dedicated memory for the AES unit to alleviate memory contention with the host processor results in Figure 9, assuming 120 pA of leakage current per SRAM bitcell [5]. In this scenario, the hybrid modes actually offer the best lifetime. The static power of the dedicated memory for the software-only mode increases to the point that the software-only lifetime is only

TABLE V: Hybrid AES Implementations

	Ctrl	AK	BS	MC	ULSNAP Perf. [Mbps]	ULSNAP t_{life} † [days]	ULSNAP Memory [B]	MSP430 Perf. [Mbps]	MSP430 t_{life} † [days]	Txrs [×1000]
0	H	H	H	H	57.02	164	40	20.05	160	180.70
1	S	H	H	H	6.05	286	40	2.05	273	73.02
2	S	H	H	S	2.27	302	592	0.49	398	58.5
3	S	H	S	H	2.35	369	1098	-	-	43.2
4	S	H	S	S	1.87	396	1650	-	-	28.6
5	S	S	H	H	2.24	323	492	-	-	44.4
6	S	S	H	S	1.39	343	1044	-	-	29.8
7	S	S	S	H	1.85	434	1550	0.11	559	14.6
8	S	S	S	S	1.54	471	2102	0.10	603	0.0

†: We assume that $1/\lambda = 1\,\text{min}$, $Y = 1.1\,\text{ms}$, $T_T = 2.2\,\text{ms}$, $\alpha = 0.1$, and that we are using a 35 mA h, 3 V CR1220 battery.

10 % better than the all-hardware configuration.

V. Conclusion

We have presented a study of AES encryption engines in the WSN design space. We developed a WSN mote battery lifetime model that accounts for the energy required to process, encrypt, and transmit collected data as well as the energy consumption during idle periods. We use this model to evaluate software, hardware, and hybrid implementations of AES on the MSP430 microprocessors as well as the ULSNAP microprocessor.

A complete QDI asynchronous transistor-level implementation of AES on ULSNAP, itself QDI, delivers 60× more throughput at 90× *less* energy per bit than the AES hardware implementation on an MSP430. The ULSNAP AES implementation also offers 30× *net* performance improvement over its software counterpart, but significantly lowers mote lifetime due to high static power consumption in the additional transistors. A software-only AES implementation can offer 3× increase in mote battery lifetime over hardware. We incorporate power gating techniques into our hybrid and full hardware designs, which reduces the gap in battery life between full software and full hardware implementations to less than 66 %. The addition of dedicated memory resources for AES increases leakage current to the point where a fully software implementation is only 10 % better than a hardware implementation, while a hybrid implementation gives 6× net performance improvement, and increases the lifetime by 10 % over its full software counterpart.

The lifetime of motes is heavily affected by the energy consumption during the idle phase of computation. While hardware implementations can offer significant performance gains, even with power gating a software-only implementation may be more appropriate depending on mote lifetime requirements. In the case where memory contention between encryption and other processor tasks is an issue, hybrid AES engines offer a lifetime improvements over an entirely software solution. Nevertheless, hybrid hardware/software AES implementations are both flexible and feasible, offering a number of options to the WSN designer.

References

[1] F. Bouesse, M. Renaudin, and F. Germain. Asynchronous AES crypto-processor including secured and optimized blocks. *JICS*, 2004.

[2] G.F. Bouesse, M. Renaudin, A. Witon, and F. Germain. A clock-less low-voltage AES crypto-processor. In *ESSCIRC*, 2005.

[3] D. Fang. Width-adaptive and non-uniform access asynchronous register files. 2004.

[4] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen. AES implementation on a grain of sand. *IEEE Information Security*, 2005.

[5] D. Ho, K. Iniewski, S. Kasnavi, A. Ivanov, and S. Natarajan. Ultra-low power 90nm 6T SRAM cell for wireless sensor network applications. In *ISCAS*, 2006.

[6] Crossbow Technology Inc. Wireless Measurement System: MICA2.

[7] Texas Instruments. CC430F6137 - MSP430 SoC With RF Core.

[8] D. Jung, T. Teixeira, and A. Savvides. Sensor node lifetime analysis: Models and tools. *ACM TOSN*, 2009.

[9] C. Karlof, N. Sastry, and D. Wagner. TinySec: A Link Layer Security Architecture for Wireless Sensor Networks. In *SenSys*. ACM, 2004.

[10] R. Karmazin, C. Ortega Otero, and R. Manohar. cellTK: Automated Layout for Asynchronous Circuits with Nonstandard Cells. In *IEEE ASYNC*, 2013.

[11] A. Lines. Pipelined asynchronous circuits. 1998.

[12] A.J. Martin. Compiling Communicating Processes for Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1986.

[13] A.J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1992.

[14] A.J. Martin, M Nystrom, and C.G. Wong. Three generations of asynchronous microprocessors. *Design & Test of Computers, IEEE*, 2003.

[15] S. Morioka and A Satoh. A 10 Gbps full-AES crypto design with a twisted-BDD S-Box architecture. In *IEEE ICCD*, 2002.

[16] L. Nazhandali, M. Minuth, and T. Austin. SenseBench: toward an accurate evaluation of sensor network processors. *IEEE IISWC*, 2005.

[17] NIST. FIPS 197: Advanced Encryption Standard. 2001.

[18] NIST. SP 800-38A. 2001.

[19] Mika Nyström, Elaine Ou, and Alain J. Martin. ROMantic: Generation and Optimization of Quasi Delay-Insensitive Read-Only Memories, 2002.

[20] I. Onat and A. Miri. An intrusion detection system for wireless sensor networks. *Wireless And Mobile Computing*, 2005.

[21] C.T.O. Otero, J. Tse, R. Karmazin, B. Hill, and R. Manohar. ULSNAP: An Ultra-low Power Event-Driven Microcontroller for Sensor Network Nodes. *IEEE ISQED*, 2014.

[22] C.T.O. Otero, J. Tse, and R. Manohar. Static Power Reduction Techniques for Asynchronous Circuits. In *IEEE ASYNC*, 2010.

[23] K.S.J. Pister, J.M. Kahn, B.E. Boser, et al. Smart dust: Wireless networks of millimeter-scale sensor nodes. *Electronics Research Laboratory Research Summary*, 1999.

[24] M. Seok, S. Hanson, Y.S. Lin, et al. The Phoenix Processor: A 30pW platform for sensor applications. In *IEEE ISVLSI*, 2008.

[25] N. Song and M.A. Perkowski. EXORCISM-MV-2: minimization of exclusive sum of products expressions for multiple-valued input incompletely specified functions. 1993.

[26] Texas Instruments. CC1101 - Low-Power Sub-1GHz RF Transceiver.

[27] Tiempo. TAES: Asynchronous AES IP datasheet.

[28] I. Verbauwhede, P. Schaumont, and H. Kuo. Design and Performance Testing of a 2.29-GB/s Rijndael Processor. *IEEE JSCC*, 2003.